

Marquette University

**e-Publications@Marquette**

---

Dissertations (1934 -)

Dissertations, Theses, and Professional  
Projects

---

## Load Balancing Algorithms for Parallel Spatial Join on HPC Platforms

Jie Yang  
*Marquette University*

Follow this and additional works at: [https://epublications.marquette.edu/dissertations\\_mu](https://epublications.marquette.edu/dissertations_mu)



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Yang, Jie, "Load Balancing Algorithms for Parallel Spatial Join on HPC Platforms" (2022). *Dissertations (1934 -)*. 1162.

[https://epublications.marquette.edu/dissertations\\_mu/1162](https://epublications.marquette.edu/dissertations_mu/1162)

LOAD BALANCING ALGORITHMS FOR PARALLEL SPATIAL JOIN  
ON HPC PLATFORMS

by  
Jie YANG, B.S.

A Dissertation submitted to the Faculty of the Graduate School,  
Marquette University,  
in Partial Fulfillment of the Requirements for  
the Degree of Doctor of Philosophy

Milwaukee, Wisconsin, USA

May 2022

© 2022

Jie YANG, B.S.

All Rights Reserved

*To my parents, my wife, and our kid.*

## ACKNOWLEDGMENTS

Jie YANG, B.S.

Marquette University, 2022

First, I would like to thank my advisor, Dr. Satish Puri, for his advice and help on my research, career, and life. His kindness, optimism, and knowledge inspire me all the time.

I would like to thank Dr. Dennis Brylow for a great internship and course experience, and for being my thesis committee member. His intelligence and deliberation always shine in my study and research life.

I would like to thank Dr. Praveen Madiraju for giving generously his time to be my thesis committee member. His kindness helps me to get over the obstacles in my research and life.

I would like to thank Dr. Hui Zhou for a great internship and invaluable feedback on my work, and for being my thesis committee member. His patience, creativity, leadership constantly guide me to climb the mountains of knowledge.

A big thanks to my Marquette colleagues, and the PMRS team at Argonne National Laboratory. I feel so lucky that I am working with so many lovely, kind, great, and optimistic people.

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

## ABSTRACT

Load Balancing Algorithms for Parallel  
Spatial Join on HPC Platforms

Jie YANG, B.S.

Marquette University, 2022

Geospatial datasets are growing in volume, complexity and heterogeneity. For efficient execution of geospatial computations and analytics on large scale datasets, parallel processing is necessary. To exploit fine-grained parallel processing on large scale compute clusters, partitioning of skewed datasets in a load-balanced way is challenging. The workload in spatial join is data dependent and highly irregular. Moreover, wide variation in the size and density of geometries from one region of the map to another, further exacerbates the load imbalance. This dissertation focuses on spatial join operation used in Geographic Information Systems (GIS) and spatial databases, where the inputs are two layers of geospatial data and the output is a combination of the two layers according to join predicate.

This dissertation introduces a novel spatial data partitioning algorithm geared towards load balancing the parallel spatial join processing. Unlike existing partitioning techniques, the proposed partitioning algorithm divides the spatial join workload instead of partitioning the individual datasets separately to provide better load-balancing. This workload partitioning algorithm has been evaluated on a high performance computing system using real-world datasets. An intermediate output-sensitive duplication avoidance technique is proposed that decreases the external memory space requirement for storing spatial join candidates across the partitions. GPU acceleration is used to further reduce the spatial partitioning runtime.

For dynamic load balancing in spatial join, a novel framework for fine-grained work stealing is presented. This framework is efficient and NUMA-aware. Performance improvements are demonstrated on shared and distributed memory architectures using threads and message passing. Experimental results show effective mitigation of data skew. The framework supports a variety of spatial join predicates and spatial overlay using partitioned and un-partitioned datasets.

**Keywords:** spatial data, spatial join, parallel computing, data partitioning, duplication avoidance, load balancing, GPU, quadtree, rtree, work-stealing, message passing interface, remote memory access

## TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . . .	i
ABSTRACT . . . . .	ii
LIST OF TABLES . . . . .	vii
LIST OF FIGURES . . . . .	viii
1 INTRODUCTION . . . . .	1
1.1 Introduction . . . . .	1
1.2 Motivation . . . . .	2
1.3 Spatial Data Partitioning and Indexing . . . . .	2
1.4 Load Balancing in Parallel Spatial Join . . . . .	4
1.4.1 Work Stealing . . . . .	5
1.4.2 NUMA . . . . .	6
1.4.3 MPI Non-blocking Communication and One-sided Communication . . . . .	6
1.5 Dissertation Statement and Contributions . . . . .	7
1.6 Thesis Outline . . . . .	9
2 LITERATURE REVIEW . . . . .	11
2.1 Introduction . . . . .	11
2.2 Geo-Spatial Data Partitioning . . . . .	13
2.2.1 Partitioning and Indexing . . . . .	13
2.2.2 Uniform Partitioning . . . . .	14
2.2.3 Quadtree Partitioning . . . . .	17
2.2.4 Multi-Jagged Partitioning . . . . .	20
2.2.5 Rectangle Tree . . . . .	24
2.2.6 Duplication Avoidance . . . . .	26

2.3	Spatial Join . . . . .	28
2.3.1	Plane Sweep . . . . .	28
2.3.2	Spatial Join using Map-Reduce . . . . .	29
2.3.3	GeoSpark . . . . .	31
2.4	Message Passing Interface . . . . .	33
2.4.1	Non-Blocking Communication . . . . .	33
2.4.2	One-Sided Communication . . . . .	35
3	EFFICIENT PARALLEL AND ADAPTIVE PARTITIONING FOR LOAD-BALANCING IN SPATIAL JOIN . . . . .	37
3.1	Introduction . . . . .	37
3.2	Background . . . . .	38
3.3	Related Work . . . . .	41
3.4	Adaptive Partitioning . . . . .	42
3.4.1	Finding candidates for partitioning . . . . .	43
3.4.2	Multithreaded Partitioning of Candidates . . . . .	45
3.5	GPU Acceleration of Adaptive Partitioning . . . . .	48
3.6	Parallel Adaptive Partitioning . . . . .	53
3.6.1	Parallel ADP for Distributed Memory . . . . .	53
3.6.2	Time Complexity . . . . .	55
3.7	Experimental Results . . . . .	56
3.7.1	Performance of Output-sensitive Duplication Avoidance . . . . .	57
3.7.2	OpenMP Quadtree Partitioning Speedup . . . . .	58
3.7.3	Computing cost for ADP . . . . .	59
3.7.4	GPU Speedup for ADP . . . . .	61
3.7.5	Weak scaling for ParADP . . . . .	62
3.7.6	Strong scaling for ParADP . . . . .	62



3.7.7	Partition Quality . . . . .	63
3.8	Conclusion . . . . .	68
3.9	Acknowledgment . . . . .	68
4	LOAD BALANCING SPATIAL JOIN BY WORK STEALING ON SHARED AND DISTRIBUTED MEMORY . . . . .	69
4.1	Introduction . . . . .	69
4.2	Background . . . . .	70
4.3	Related Work . . . . .	72
4.3.1	Spatial Join . . . . .	72
4.3.2	Load Balancing in Parallel Spatial Join . . . . .	73
4.3.3	Work Stealing . . . . .	73
4.3.4	NUMA . . . . .	74
4.3.5	RMA and MPI Non-blocking Communication . . . . .	74
4.4	Implementation of Work Stealing Spatial Join . . . . .	75
4.4.1	Work Stealing Queue . . . . .	75
4.4.2	NUMA Memory Policies . . . . .	75
4.4.3	Algorithm . . . . .	77
4.4.4	Handling Partitioned and Un-Partitioned Datasets . . . . .	79
4.5	Framework of Work Stealing Spatial Join on Distributed Memory . . . . .	80
4.5.1	Overall Framework . . . . .	80
4.5.2	Worker Threads . . . . .	81
4.5.3	Coordinator in Send Status . . . . .	82
4.5.4	Coordinator in Receive Status . . . . .	82
4.5.5	Internode Communication . . . . .	83
4.5.6	Theoretical Analysis . . . . .	84
4.6	Experimental Results . . . . .	85

4.6.1	NUMA . . . . .	87
4.6.2	Tasks Composition of WSSJ . . . . .	88
4.6.3	Tasks Composition of WSSJ-DM . . . . .	89
4.6.4	Comparison Experiments for WSSJ . . . . .	90
4.6.5	Comparison Experiments for WSSJ-DM . . . . .	91
4.6.6	Strong Scaling for WSSJ-DM . . . . .	92
4.6.7	Benchmark . . . . .	92
4.7	Conclusion . . . . .	95
5	ASYNCHRONOUS DYNAMIC LOAD BALANCING BASED SPATIAL JOIN	97
5.1	Introduction . . . . .	97
5.2	Spatial Data Computing Costs Modeling . . . . .	98
5.3	Spatial Data Partitioning . . . . .	98
5.4	Dynamic Load-balancing . . . . .	101
6	SUMMARY AND FUTURE WORK . . . . .	102
6.1	Summary . . . . .	102
6.2	Future Work . . . . .	103
	Bibliography . . . . .	104
	Appendices . . . . .	113
A.1	Pseudo code for CUDA quadtree partition . . . . .	113

## LIST OF TABLES

3.1	Attributes of the data sets . . . . .	57
3.2	OpenMP based Quadtree Partitioning time (seconds) . . . . .	59
3.3	ADP total and break-down execution time for different pairs of datasets . .	60
3.4	ADP-GPU execution time for different pairs of datasets on an Nvidia Titan-V	61
3.5	ParADP execution time for weak scaling . . . . .	62
3.6	ParADP execution time for strong scaling . . . . .	63
4.1	Attributes of the datasets . . . . .	86
4.2	Execution time (in sec) for Sequential Indexed Spatial Join, WSSJ (36 cores), WSSJ-DM (25 nodes) performing spatial join on different pairs of un-partitioned datasets. . . . .	95

## LIST OF FIGURES

1.1	Spatial Join techniques. . . . .	3
2.1	If both $R$ and $S$ are partitioned using a same grid, then the join process only needs to consider corresponding cells. . . . .	15
2.2	Quadtree visuallization . . . . .	18
2.3	An example of the region quadtree. (Originally presented by Hanan Samet). . . . .	21
2.4	Examples of using RCB and MJ. The cutting order is either indicated by colors (red $\rightarrow$ blue $\rightarrow$ green $\rightarrow$ pink $\rightarrow$ orange) or by the thickness of the cutting line (thickest to thinnest). The two MJ variants produce similar partitions, but MJ with migration allows more concurrency during partitioning. (Originally presented by Mehmet Deveci) . . . . .	22
2.5	Rtree visuallization. . . . .	25
2.6	An example of using <i>reference point method</i> . . . . .	27
2.7	An overview of the mapreduce paradigm. (Originally presented by Jeffrey Dean). . . . .	30
2.8	An overview of the GeoSpark framework. (Originally presented by Jia Yu). . . . .	32
2.9	An example (broadcasting) of using MPI non-blocking APIs. . . . .	34
2.10	An example of active MPI one-sided communication. . . . .	36
3.1	Number of geometries shown in each grid cell. The workload of a cell in grid C is the product of the number of geometries present in corresponding cells in A and B (e.g., workload in the fourth cell is $9*5$ ). . . . .	39
3.2	Mapping of candidates to grid cells. $(r_1, s_1)$ , $(r_1, s_2)$ , $(r_2, s_3)$ are candidates. Due to our output-sensitive method, geometry $r_1$ is not stored in cell IDs (D, I), (C, I), (B, III), (B, IV), and (A, IV) even though it passes through these grid cells. Instead, $r_1$ is stored in cells (C, II) and (A, V) because it is part of two candidates $(r_1, s_1)$ and $(r_1, s_2)$ only. . . . .	44
3.3	Visualization of Algorithm 6 and 7 . . . . .	49
3.4	ParADP using 4 compute nodes with 4 cores in each node. Longitudinal thick black lines are generated first for rearranging data based on its stripe boundary in each node. The green lines are generated by each node independently. Every CPU core/thread is assigned one cell. The thin red lines are partitioning boundaries generated by each CPU core individually. . . . .	55

3.5	Parallel partitioning of the <i>roads</i> and the <i>parks</i> into 8192 grid cells using ParADP . . . . .	56
3.6	Storage space needed using different partitioning techniques . . . . .	57
3.7	Max process time and min process time for GEOS <i>Intersects</i> method using <i>parks</i> and <i>sports</i> using an increasing number of grid cells generated by ADP. 84 cores are used for all cases. . . . .	59
3.8	Speedups of ParADP w.r.t. ADP for generating a grid with 65536 cells using two datasets - 1) <i>roads</i> (72 million polylines) and 2) <i>parks</i> (10 million polygons). . . . .	63
3.9	Box-plot showing distribution of execution time by different MPI processes running GEOS <i>Intersects</i> query using <i>parks</i> and the <i>sports</i> data. Each time the data sets are partitioned into 8192 parts. Max process execution time along with few outliers are also shown for each partitioning scheme. . . . .	64
3.10	Maximum and minimum process execution times. Datasets <i>parks</i> and <i>sports</i> were used and partitioned into 8192 parts. . . . .	65
3.11	Execution time of applying <i>Intersects</i> on different cells of the partitioned <i>parks</i> and <i>sports</i> . The data sets are partitioned into 8192 cells by ParADP and Quadtree partitioning. . . . .	66
3.12	Execution time of applying <i>Intersects</i> on different cells of the partitioned <i>parks</i> and <i>sports</i> data. The data sets are partitioned into 8192 cells by ParADP and ADP. . . . .	66
4.1	The Work Stealing Spatial Join model in shared memory (WSSJ). The blue arrows show the direction of flows of Spatial Join tasks within a worker thread. The red arrows show the direction of flows for stolen tasks. . . . .	76
4.2	The Work Stealing Spatial Join system on distributed memory (WSSJ-DM). The solid blue arrows show the directions of the flows of spatial join tasks. The dashed orange arrows show the directions of the flows of control messages. “NB send/rcv” stands for “MPI non-blocking send or receive”. “Gen buf” stands for “Generate send buffer” and “Parse received buffer”. . . . .	81
4.3	Theoretical performance modeling of WSSJ-DM before reaching bottleneck. . . . .	86
4.4	Execution time comparison of different NUMA policies in WSSJ for performing <i>ST_Intersects</i> on <i>Sports</i> and <i>Cemetery</i> . . . . .	88
4.5	Composition of tasks at different WSSJ workers. Both cases used 36 workers to perform <i>ST_Intersection</i> on <i>Lakes</i> and <i>Parks</i> . . . . .	89
4.6	Composition of tasks at different WSSJ-DM nodes. Both cases used 5 nodes to perform <i>ST_Intersection</i> on <i>Lakes</i> and <i>Parks</i> . . . . .	90

4.7	Execution time comparison of using WSSJ, Master-Worker (MW), and Round Robin assignment (R-R) to perform different spatial joins on <i>Lakes</i> and <i>Sports</i> , which are spatial partitioned into 8192 sub-sets using ADP partitioning. . . . .	91
4.8	Execution time comparison among WSSJ-DM, ADLB, SPINOJA-DM, and Round Robin assignment (R-R) performing spatial joins on <i>Lakes</i> and <i>Parks</i> , which are spatially partitioned into 8192 grid cells using ADP partitioning. . . . .	93
4.9	Execution time and speedup plot of WSSJ-DM w.r.t sequential join. For comparison, <i>ST_INTERSECTION</i> was used on <i>Lakes</i> and <i>Parks</i> . Input data was partitioned into 8192 grid cells using different approaches. . . . .	94
5.1	The execution time of GEOS to find the intersections between two geometries. In (a) all geometry collections are kept; in (b) all geometry collections are broken down into single geometries in the samples. . . . .	99
5.2	Max and min processing times of MPI processes to perform join on two spatial datasets, lakes (8.4 M polygons) and sports (1.8 M polygons). . . .	100
5.3	Execution time comparison between two versions: 1) Using ADLB and 2) MPI-GIS dynamic load-balancing for join between Sports (1.8 M polygons) and Roads (72 M polylines). . . . .	101

## CHAPTER 1

### INTRODUCTION

#### 1.1 Introduction

We are in the era of Spatial Big Data. Due to the developments of topographic techniques, clear satellite imagery, and various means for collecting information, geospatial datasets are growing in volume, complexity and heterogeneity. For example, OpenStreetMap data for the whole world is about 1 terabyte <sup>1</sup> and NASA world climate datasets are about 17 terabytes <sup>2</sup>. Processing such large data and running spatial analytics require a lot of time. Spatial data volume and variety makes processing and analytics both data-intensive and compute-intensive tasks.

Analyzing spatial data is important in many fields and leading to new discoveries. There are frequent occurrences of natural disasters such as earthquakes, floods, wildfires. Governments and researchers analyze spatial data related to those disasters to predict, control damage, and rescue victims [1]. Spatial data can be used to locate the center(s) of contagions like plague, COVID-19 and help to prevent them [2,3]. Web users contribute millions of posts with geographic tags on social platforms per day. Those data can be used to study human behaviors, research on commercial purposes, and so on [4]. Urban planning directly relies on analyzing spatial data to decide locations of vital public infrastructures such as hospitals, airports and even roads [5]. The need of analyzing spatial data is fundamental in research fields such as geology [6].

Spatial join is an important collection of operations for analyzing spatial data. It is used to find the relations between multiple spatial objects. With the increasing volume and complexity of spatial data, there is an increasing demand for efficient spatial join techniques. Parallel programming is a plain solution to accelerate the process of spatial join. However, as the size and density of geometries varies from one region of the map to another [7, 8],

---

<sup>1</sup><https://wiki.openstreetmap.org/wiki/Planet.osm>

<sup>2</sup><https://cds.nccs.nasa.gov/nex>

data skew and load imbalance are vital issues for parallel spatial join [9]. To make better utilizations of modern computing architectures (multicore CPUs, GPUs), solutions to the load balancing problem are important.

## 1.2 Motivation

With the evolution of the computing architectures, modern High Performance Clusters (HPC) can provide massive computing resources. However, when we use HPCs to process spatial join problems, the load imbalance in parallel spatial join wastes the majority of HPCs' computing ability.

Due to the developments of topographic techniques, clear satellite imagery, and various means for collecting information, geospatial datasets are growing in volume, complexity, and heterogeneity. For efficient execution of spatial computations and analytics on large spatial data sets, parallel processing is required. To exploit fine-grained parallel processing in large scale compute clusters, partitioning in a load-balanced way is necessary for skewed datasets.

Load imbalance due to data skew limits the scalability of parallel spatial join. There are some techniques to address this problem. One of the techniques is to use data and space partitioning to minimize workload differences across threads/processes. However, load imbalance still exists due to differences in join costs of different pairs of input geometries in the partitions. Another technique is to share spatial join tasks among threads using a shared queue.

## 1.3 Spatial Data Partitioning and Indexing

In Geographic Information System (GIS) and spatial database, two datasets are combined based on some spatial relationship among geometries in the input datasets.

Spatial join [10] involves two spatial datasets  $R$  and  $S$ . Spatial join operations can be classified into two types: type 1 is to determine spatial relationship<sup>3</sup>, such as *ST\_Within*,

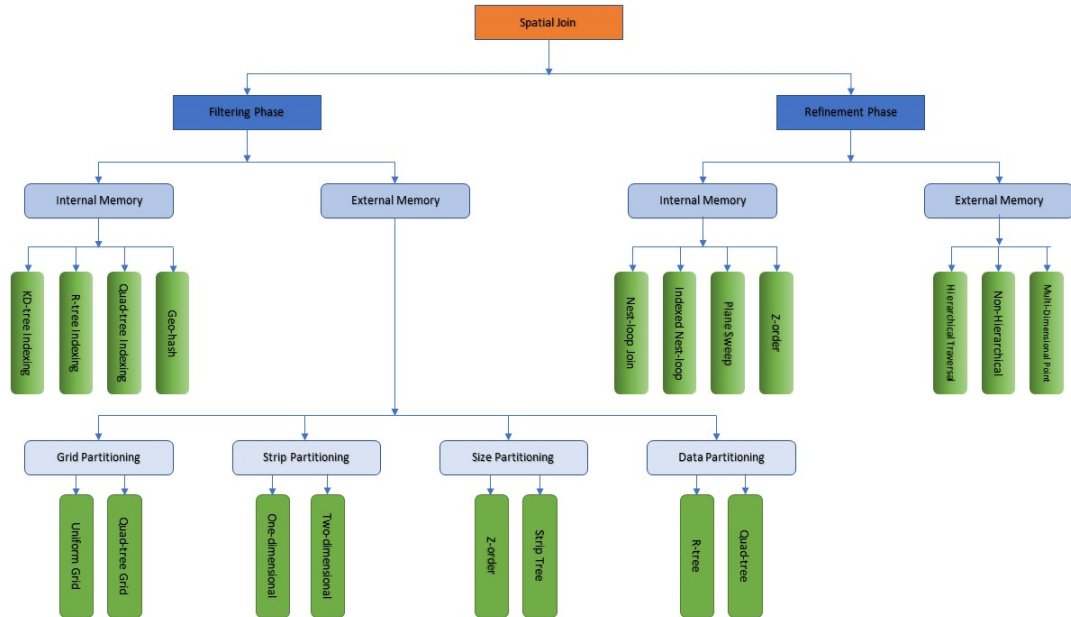
---

<sup>3</sup>[https://postgis.net/docs/reference.html#Spatial\\_Relationships](https://postgis.net/docs/reference.html#Spatial_Relationships)



*ST\_Intersects*, and other operations; type 2 is to compute overlay area<sup>4</sup>, such as *ST\_Intersection*, *ST\_Union*, and other operations. *ST\_Intersects* is used to answer a query — Is there any overlap between the two geometries? This operation is often faster because the computation can stop as soon as the spatial relationship is confirmed. Join operation of the second type is used in map overlay, and it is more expensive because the entire output geometries have to be computed.

A spatial join on two datasets can be performed in two phases: 1) filtering phase and 2) refinement phase. For techniques working the filtering phase, the minimum bounding rectangles (MBR) of geometries instead of the geometries themselves are usually used to perform efficient filtering. A collection of candidate pairs is produced and passed to the refinement phase. The refinement phase then removes false positives and produces a set of pairs in relationship join or a set of new geometries in overlay join.



**FIGURE 1.1** Spatial Join techniques.

Figure 1.1 provides an overview of current spatial join techniques. Besides for the filtering or refinement phase, being suitable for Internal memory or external memory can be another classification for spatial join techniques. External memory solutions usually aim

<sup>4</sup>[https://postgis.net/docs/reference.html#Overlay\\_Functions](https://postgis.net/docs/reference.html#Overlay_Functions)

at large spatial data-sets which can not be loaded on a single machine.

*Spatial Join operations costs for a pair of geometries:* assume the numbers of vertices are  $n$  and  $m$ . In the filtering phase, the MBRs of two geometries are used to verify if two geometries potentially intersect with each other. This computing is a constant cost as MBRs are four-point rectangles. The numbers of vertices of geometries do not affect this cost. In the refinement phase, the cost varies by join operations and algorithms. Take *ST-Intersection* as an example: a naive line-by-line algorithm can take  $O((n + m)^2)$ ; Plane Sweeping [11], one of the best spatial join algorithms, takes  $O((n + m + k)\log(n + m))$ , where  $k$  is the number of vertices of the intersection.

#### 1.4 Load Balancing in Parallel Spatial Join

Parallel spatial join solutions utilizes multicore CPUs in shared memory, or multiple computing nodes in distributed memory. However, due to data skew and join operation cost uncertainty, load unbalancing is a critical problem for recent parallel spatial join solutions.

Various techniques have been developed to reduce load unbalancing in parallel spatial join. Spatial join parallelization has been discussed in [9, 12–17]. [9] uses object decomposition based declustering to mitigate data processing skew on shared memory. A task manager is used to arranging join tasks among workers for load balancing. [12] divides parallel spatial join into three phases: 1) Partitioning Phase, 2) Join Phase and 3) Refinement Phase. Two parallel spatial join algorithms Clone JOIN and Shadow Join was introduced in [12]. [12] uses partitioning based strategies to decluster spatial data for parallel spatial join. [13] builds two hash tables on two inputs for non-blocking parallel spatial join, with optional duplication avoidance. The MapReduce paradigm is used in Spatial Join with MapReduce [14], a parallel spatial join algorithm using no spatial indexes, with a tile-based duplication avoidance. [15] presented a parallel spatial join framework based on MPI. The framework uses a prediction model to migrate files among MPI processes for load balancing. [16] uses bitmaps to determine the number of spatial objects to perform dynamic load balancing. Spatial join was implemented on hypercube architecture of the Connec-

tion Machine [17] using Census TIGER/Line data. [17] noted that in parallel spatial join, the execution time increases by increasing the capacity of a Rtree node, while the execution time decreases with a higher PMR (Polygonal Map Random) quadtree bucket capacity. Declustering is proposed as a load balancing strategy in [18, 19]. [18] studied the impact of declustering spatial data for shared-nothing parallel structures by implementing a round-robin and a top-left declustering strategies, with or without redundancy. [19] parallelized spatial GIS range-query (one spatial join operation) by partitioned spatial data. The authors also compared the impacts of static schedule methods and dynamic load balancing methods over parallel spatial join, and proved that the hierarchical declustering approach provides better load balancing. [20] introduced PBSM (Partition Based Spatial-Merge) by using partitioning of universe into tiles (grid cells). The tiles are then assigned to processors in a round-robin fashion. The experiments show that PBSM has better performance than R-tree based spatial join and indexed nested loops spatial join. [21] performs parallel multidimensional similarity join using Partitioning based Quantiling (PQ) algorithms. The paper proves that considering the number of join tests is beneficial for load balancing.

#### 1.4.1 Work Stealing

Work stealing is a dynamic load balancing strategy [22–26]. It has been used in shared memory and distributed memory [23] load balancing solutions.

Chase-Lev’s lock-free deque [24] is an important data structure in many shared-memory work stealing designs. The deque uses a *dynamic-cyclic-array*, which allows: 1) the owner to push and pop elements from the top of the deque, 2) others to perform concurrent lock-free steal from the bottom of the deque. Nhat’s Work Stealing Queue [25] implementation in C++11 is based on Chase-Lev’s lock-free deque and shows a remarkable performance in benchmarks. We use it in our work stealing implementations. For simplicity, we have referred to Work Stealing Queue as queue.

### 1.4.2 NUMA

The execution of spatial join computations are impacted by NUMA memory policies. Spatial join algorithms allocate a temporary buffer to carry out intermediate steps of join algorithm on two geometries. The spatial objects are copied to the temporary buffer to carry out Quadtree partitioning of an individual geometry, to order the coordinates, and to populate the intersection matrix.

The default NUMA policy on most Linux systems after boot-up is *MPOL\_DEFAULT*, which is “local allocation”. Under this policy, Linux will attempt to satisfy memory requests from the nearest NUMA node of the CPU which submits the memory requests. *MPOL\_DEFAULT* works fine in many scenarios. However, in terms of work stealing, a thread on one NUMA node can steal a task from another NUMA node. A page in memory is accessed by multiple threads. For spatial join, in all experiments we have conducted so far, the tasks on a few worker threads (usually 1 to 4) take much longer to finish than the rest of the threads. When multiple threads allocate and write to temporary buffers for tasks from remote NUMA nodes, there can be memory requests congestion.

*MPOL\_BIND* and *MPOL\_PREFERRED* can mitigate the memory requests congestion issue. Under these two policies, the temporary buffers are on the same NUMA node as the pairs of geometries to be joined. The issue with *MPOL\_BIND* is that it is a strict policy; the OS can only utilize the memory on specified NUMA node(s).

Under *MPOL\_INTERLEAVE* mode, the memory allocations are uniformly distributed among all NUMA nodes. The temporary buffers are allocated in an interleaved manner as the pairs of geometries are joined.

### 1.4.3 MPI Non-blocking Communication and One-sided Communication

We have used one-sided (put/get) Message Passing Interface (MPI) functions for communicating data among cooperating. One-sided programming model is referred to as Remote Memory Access (RMA) in MPI. It is suitable for expressing irregular communication patterns that arise while coordinating tasks among processes in distributed memory [27].

Non-blocking MPI functions can be leveraged to overlap communication operations with computational steps of spatial join.

## 1.5 Dissertation Statement and Contributions

The goal of this dissertation is to solve the load imbalance problems in parallel spatial join. For that purpose, this dissertation focuses on spatial data management, storage methods and spatial join tasks scheduling, sharing techniques to mitigate the loading imbalance problems. This dissertation focuses on improving the utilization of many-core architecture in the field of spatial data processing.

### Dissertation Statement

*Parallel spatial join is a promised analyzing tool for spatial data. Spatial data partitioning methods and workload schedulers are important for parallel spatial join programs to utilize modern many core architectures.*

To address the load imbalance problems in current parallel spatial join approaches, this thesis introduces a novel spatial data partitioning approach, and a distributed work stealing system for spatial join.

**An efficient parallel and adaptive partitioning method:** Our partitioning method for spatial join uses Adaptive Partitioning (ADP) technique, which is based on Quadtree partitioning. Unlike existing partitioning techniques, ADP partitions the spatial join workload instead of partitioning the individual datasets separately to provide better load-balancing. Based on our experimental evaluation, ADP partitions spatial data in a more balanced way than Quadtree partitioning and Uniform grid partitioning. ADP uses an output-sensitive duplication avoidance technique which minimizes duplication of geometries that are not part of spatial join output. In a distributed memory environment, this technique can reduce data communication and storage requirements compared to traditional methods.

**A work stealing spatial join framework on shared and distributed memory:** we present our parallel spatial join system, WSSJ-DM. WSSJ-DM benefits from balanced par-

tioning research. Moreover, we experimentally show that our system works well with unbalanced partitioning and spatially un-partitioned datasets, with minor impact on its overall performance. WSSJ-DM uses work stealing technique to share join tasks on shared memory. We study the effect of memory affinity in work stealing operations involved in spatial join in a NUMA-aware system. On distributed memory, non-blocking communications are used to shuffle tasks between busy and idle nodes.

These two contributions show the following key intellectual insights:

- 1) We show that ADP provides better load balancing per partition than existing approaches. ADP considers the workload of the refinement phase of each partition, and always cuts the partition with most workload.
- 2) While being partitioning into more parts, more external memory space is needed to store the partitioning result due to duplications. By eliminating unnecessary duplications and objects not participating join, ADP is able to use significant smaller external memory space than current approaches.
- 3) ADP can be extended to support multiple many-core architectures such as multicore machine, distributed cluster, and GPU. The parallelized ADP shows a good scale and can partition large datasets in a short time.
- 4) We demonstrate effective mitigation of data skew in a fine-grained manner to avoid stragglers (threads taking much longer than others to finish). Both WSSJ and WSSJ-DM are proved to be load balancing and efficient.
- 5) WSSJ implemented using non-blocking task queues solves the NP-complete problem - load imbalance problem in parallel spatial join - for typical spatial join cases in shared memory.
- 6) WSSJ-DM overlaps communication and computation parts of parallel spatial join, which makes it is the first application that solves the intersection problem on two 10-GB datasets in 30 seconds.

## 1.6 Thesis Outline

This section outlines the thesis and briefly introduces the rest chapters.

Chapter 2 presents literature review on the topics of this thesis. Each of the subsequent chapters presents a solution or technique which directly related or can be used for parallel spatial join. The techniques, problems, solutions and related work are introduced in each section.

Chapter 3 presents Adaptive Partitioning [28], a novel spatial data partitioning algorithm. ADP takes the distribution of geometries in both layers into consideration which can improve spatial partitioning by producing grid cells with similar workload. Since we use a filtering-based approach to find the potentially overlapping geometries, we can minimize duplication of geometries that do not take part in spatial computations in the refine phase. Moreover, we parallize ADP on multiple tools/platforms, including Pthread, MPI, CUDA, and OpenMP. The MPI + Pthread implementation (ParADP) scales up to four thousands CPU cores.

Chapter 4 presents our parallel spatial join system, WSSJ-DM. WSSJ-DM benefits from balanced partitioning research. Moreover, we experimentally show that our system works well with unbalanced partitioning and spatially un-partitioned datasets, with minor impact on its overall performance. WSSJ-DM uses work stealing technique to share join tasks on shared memory. We study the effect of memory affinity in work stealing operations involved in spatial join in a NUMA-aware system. On distributed memory, non-blocking communications are used to shuffle tasks between busy and idle nodes.

Chapter 5 presents the preliminary work for Chapter 3 and Chapter 4. It presents spatial data partitioning techniques such as quadtree and uniform grid partitioning based on modeling of spatial join [10] cost. In addition, we present Asynchronous Dynamic Load Balancing (ADLB) [26] based spatial join implementation. The spatial join times modeling experiments expound how geometry collections make load-balancing difficult. We use different spatial data partitioning techniques to find a more balanced method. We evaluate the performance of ADLB-based program by comparing with another MPI-GIS [29]

implementation.

Chapter 6 summarizes this thesis and discusses future research.



## CHAPTER 2

### LITERATURE REVIEW

#### 2.1 Introduction

With the increasing volume and complexity of spatial data, spatial data partitioning techniques are significant to handle mega size spatial datasets. Spatial data partitioning not only benefits parallel spatial join, but also benefits sequential join because of domain decomposition. Additionally, partitioned spatial data is easy to be stored and loaded to the internal memory due to the independence of sub datasets.

Partitioning spatial data has been well-studied in literature. To begin with, [10] introduced a naive approach of partitioning spatial data and utilizing the partitioned data for spatial join. For example, a two-dimensional coordinate system is partitioned into four Quadrants by the x-axis and y-axis; a three-dimensional space can be further partitioned into 8 subspaces if z-axis is used. Two original datasets,  $R$  and  $S$ , are partitioned using the same grid and their subsets can be stored in external memory. The corresponding subsets in  $R$  and  $S$  can be loaded into internal memory for spatial join. Those corresponding pairs are independent of each other, so a shared-nothing parallel spatial join framework can be easily implemented by partitioned spatial data. However, if a naive partitioning approach is used such as the Uniform Partitioning, the load imbalance problem strictly limits the parallel spatial join framework's performance. The choice of partitioning scheme depends on the application where it is used.

The outline of literature review for spatial data partitioning is listed as following:

- 1) A basic introduction of spatial data partitioning is introduced in Section 2.2.1;
- 2) The uniform partitioning is introduced in Section 2.2.2;
- 3) A paper about quadtree is introduced in Section 2.2.3;
- 4) The rectangle tree data structure and related papers are introduced in Section 2.2.5;
- 5) The multi-jagged partitioning technique is introduced in 2.2.4;

6) Finally, an inline duplication avoidance technique is introduced in 2.2.6.

Spatial join is a collection of important operations for analyzing spatial data. In spatial analytics, combining two or more datasets gives us insights that are not available in a single dataset. Spatial Join operations can be divided into two types: 1) join operations to find spatial relationships, such as intersects, within, and other operations; 2) join operations to compute overlay area, such as intersection, union, and other operations. A spatial join on two datasets can be performed in two phases: 1) filtering phase and 2) refinement phase. In the filtering phase, the minimum bounding rectangles (MBR) of geometries are used to produce a collection of candidate pairs, in which MBRs of two geometries from two datasets overlap.

The outline of literature review for spatial join is listed as following:

- 1) The Plane Sweep technique is introduced in Section 2.3.1;
- 2) A parallel spatial join framework derived from Map-Reduce is introduced in Section 2.3.2;

The Message Passing Interface (MPI) is the de-facto message passing standard on High Performance Computing platforms. An important function provided by MPI is to allow processes on a cluster to communicate with each other in various methods and formats. For instance, *MPI\_Bcast* allows one process to broadcast its data with other processes. The MPI standard only defines the logic of MPI operations. An implementation of MPI may have its own efficient way to design functions such as *MPI\_Bcast*. The utilization of MPI is a key point of designing an efficient HPC program.

The outline of literature review for Message Passing Interface (MPI) related techniques is listed as following:

- 1) MPI non-blocking communication is introduced in Section 2.4.1;
- 2) MPI one-sided communication is introduced in Section 2.4.2.

## 2.2 Geo-Spatial Data Partitioning

### 2.2.1 Partitioning and Indexing

Spatial data usually can be partitioned based on a grid, such as uniform partitioning, or on the data itself, such as partitioning using a Rectangle-tree (R-tree) index or a quadtree index, etc. To illustrate how spatial data partitioning works, Edwin Jacox and Hanan Samet introduce a simplified algorithm in [10], as shown in Algorithm 1. The basic idea is to define a customized grid and use it to partition both datasets,  $R$  and  $S$ . The number of cells in the grid is mainly evaluated by the available memory, that is, to have all sub-sets to fit in the memory. Every spatial object is put into each partition which overlaps with it. If an object is overlapped by multiple partitions, it will be included in all of those partitions. An example is shown in Figure 2.1, there is a rectangle laying on the board of partition 1 and 2. This rectangle will be included in both partition  $R1$  and  $R2$ . Any future spatial join programs which use  $R1$  or  $R2$  have to include that rectangle. And this brings in the duplication problem in parallel spatial join, which is introduced in Section 2.2.6. In Figure 2.1,  $R$  and  $S$  are partitioned into four cells using the same grid. Each cell in  $R$  will only be joined with the corresponding cell in  $S$ . For instance,  $R1$  only joins with  $S1$ , and there is no need to consider joining  $R1$  with  $S2$ .

The first step in Algorithm 1 is to calculate how many partitions are needed. The purpose of this step is to ensure that each pair of two corresponding partitions is small enough to fit in internal memory. The available internal memory and the size of Minimum Bounding Rectangle (MBR) are used by the function **DETERMINE\_PARTITIONS** to calculate the minimal number of partitions. However, this *min\_num\_of\_partitions* is usually incorrect due to the skew of spatial data. More advanced partitioning algorithms such as quadtree Partitioning in Section 2.2.3 and Multi-Jagged Partitioning in Section 2.2.4 and others can handle this problem by partitioning more on dense areas (area with more spatial objects). *Min\_num\_of\_partitions* is used to get the list of partition cells by function **DETERMINE\_PARTITIONS**.

The second step is to use *partition\_list* partitioning  $R$  and  $S$ . Once *partition\_list* is gen-

erated, the next step is to scan  $R$  and  $S$ , and place every object into its corresponding partition(s). The partitions are then stored the partitioning results to external memory and can be reused. Finally, in the join phase, each partition in  $R$  and its corresponding partition in  $S$  are loaded to the internal memory and joined. Detailed Spatial Join techniques are introduced in Section 2.3.

---

**Algorithm 1** Grid Join
 

---

```

1: Input: Spatial data sets  $R$  and  $S$ .
2: Output: Join results  $RESULTS$ .
3: /* calculate the number of cells in the grid */
4:  $m \leftarrow$  available internal memory
5:  $MBR\_size \leftarrow$  space needed to store one MBR
6:  $min\_num\_of\_partitions \leftarrow (sizeof(R) + sizeof(S)) * MBR\_size / m$ 
7:  $partition\_list \leftarrow \mathbf{DETERMINE\_PARTITIONS}(min\_num\_of\_partitions)$ 
8: /* partition data and store it to external memory */
9:  $partition\_results\_R \leftarrow \mathbf{PARTITION\_DATA}(partition\_list, R)$ 
10:  $partition\_results\_S \leftarrow \mathbf{PARTITION\_DATA}(partition\_list, S)$ 
11: /* perform join operation on partitioned data */
12: for each  $partition$  in  $partition\_list$  do
13:    $partition\_R \leftarrow \mathbf{READ\_PARTITION}(partition\_results\_R)$ 
14:    $partition\_S \leftarrow \mathbf{READ\_PARTITION}(partition\_results\_S)$ 
15:    $RESULTS \leftarrow RESULTS \cup \mathbf{SPATIAL\_JOIN}(partition\_R, partition\_S)$ 
16: end for

```

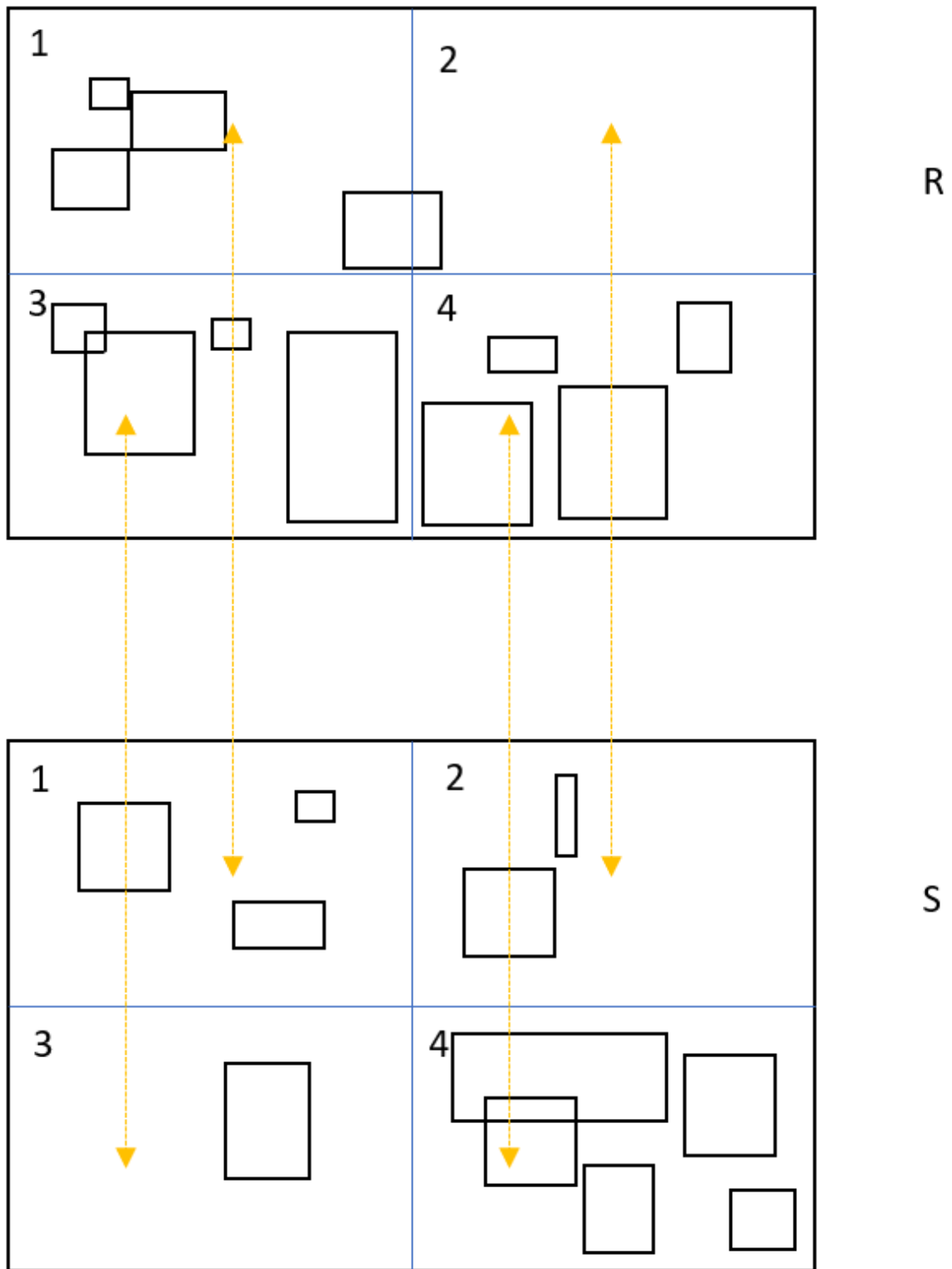
---

### 2.2.2 Uniform Partitioning

Uniform partitioning is simple and is already shown in Figure 2.1. For instance, a map is partitioned uniformly by the longitude and latitude lines (not physically but numerically). Though uniform partitioning is simple, it is good to be used as a starting point as in [20].

Jignesh Patel and David DeWitt use Partition Based Spatial-Merge (PBSM) to perform spatial join in [20]. PBSM does not require any indices on the joining attribute of the two inputs,  $R$  and  $S$ . Instead, it uses the unique identifiers of each geometry in  $R$  and  $S$  to map them in the filter step.

- 1) **Filter Step:** PBSM loads all geometries from  $R$  and  $S$  and store them as *key-pointer elements*,  $R^{(kp)}$  or  $S^{(kp)}$ . With  $R^{(kp)}$  and  $S^{(kp)}$ , PBSM performs a *Plane-sweep* opera-



**FIGURE 2.1** If both  $R$  and  $S$  are partitioned using a same grid, then the join process only needs to consider corresponding cells.

tion, which is introduced in 2.2.6, to find all pairs of intersecting Minimum Bounding Rectangles (MBRs) between  $R$  and  $S$ . If  $R^{(kp)}$  and  $S^{(kp)}$  are too large for the internal

memory, they can be partitioned into non-disjoint subsets to be fitted in the memory. The partitioning approach is uniform and similar to the algorithm described in Section 2.2.1. The duplication mechanism is also same. In this step, geometries from  $R$  and  $S$  are mapped to each other if their MBRs are intersected. The pairs are stored in a fashion that geometries from  $R$  are used as the keys and those from  $S$  are the values. This *key-value* structure can avoid random seeking in  $R$  and  $S$ .

- 2) **Refinement Step:** The *key-value* pairs are loaded in a manner that loads enough keys (geometries from  $R$ ) fitting the internal memory, with their corresponding values (geometries from  $S$ ). The geometries from  $S$  are loaded sequentially into the memory. Then the join operation is performed, and the outputs are recorded based on the unique identifiers of each geometry.

There are several steps to carry out these two steps. First the number of partitions is needed to be decided. As the plane-sweep requires all data to be in the internal memory, PBSM uses the cardinalities of  $R$  and  $S$ , the size of the internal memory  $M$ , and the size of a *key-value* pointer. The partition number  $P$  is decided by the following formula:

$$P = \left\lceil \frac{(\text{sizeof}(R) + \text{sizeof}(S)) * \text{sizeof}(\text{key} - \text{value pointer})}{M} \right\rceil$$

As being stated, the partitioning function used by PBSM is the Uniform Partitioning. To mitigate the skew of the Uniform Partitioning, the entire universe is partitioned into more tiles than partitions and each tile is mapped to a partition in a round-robin or hashing scheme. It is studied in [20] that with more tiles the data skew is reduced, but the replication overhead is higher. PBSM can also choose to dynamically partition large partitions to handle skew, which becomes a quadtree like partitioning in 2.2.3.

Additionally, this paper contributes a comprehensive performance experiments for three different spatial join algorithms:

- 1) an indexed nested loops based spatial join algorithm,
- 2) an R-tree based spatial join algorithm,

3) and the PBSM algorithm.

The indexed nested loops based spatial join algorithm first builds an index on the smaller set between  $R$  and  $S$ . An Rtree index is typically used. Then the inputs are sorted by turing MBRs into Hilbert values and this makes the geometries to be joint spacially close. The R-tree based spatial join algorithm first builds an index on both  $R$  and  $S$ . The two R-trees allows fast synchronous depth-first traversal to find all matching pairs between  $R$  and  $S$ .

With spatial datasets from the real world, these comparing experiments carry out two important conclusions:

- 1) the PBSM algorithm is more effective than the other two without exisiting index,
- 2) and prove that loading an index is cheaper than constructing one at run-time.

As a conclusion, Partition Based Spatial-Merge does not need and indices, which makes it sutiable for situations like intermediate computations. PBSM can handle spatial datasets which larger than available internal memory by partitioning and mapping.

### 2.2.3 Quadtree Partitioning

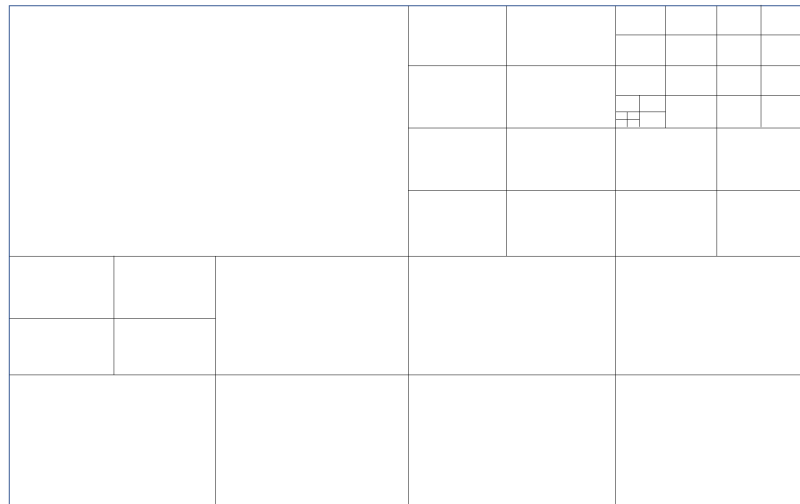
Quadtree Partitioning keeps partitioning the largest cell of all its subsets until the target number of partition is meet. It is introduced in [30, 31] as a tree data structure for spatial data, and we can also use it as a partitioning algorithm. The main idea is to overlay a tree of coordinates on top of the hierarchical representation of an intdgrated circuit. The coordinate tree enables one to find quickly the set of all objects that intersect a given window. When a quadtree being visuallized, it can be either in a two-dimensional style in Figure 2.2(a) or in an abstract tree in Figure 2.2(b). Due to the limit of usable area, the abstract quadtree in Figure 2.2(b) is only partially corresponding to the quadtree in Figure 2.2(a).

Quadtree is a hierachical data structure whose space is recursively partitioned into four children of same size. The following features make diffrent types of quadtree:

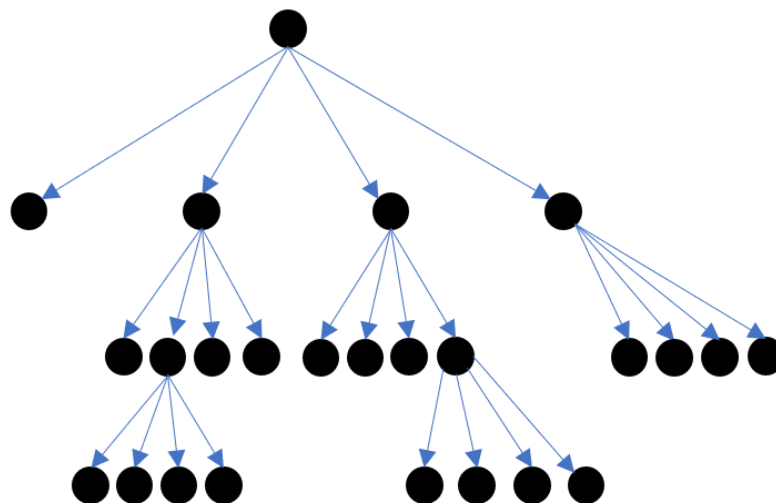
- 1) the type of data the quadtree is used to represent, such as geographical data;
- 2) the principle guiding the decomposition process, such as parallel or sequential;

3) the data is in variable form or not.

A quadtree can be used to represent multiple types of objects such as points, rectangles, regions, volumes, or even irregular geometries which can be presented by rectangles. A regular decomposition can be happened on each level and partition each parts equally, or it can be input-sensitive. The depth (which also means how many times a quadtree can be partitioned) can be determined by the input, or be fixed initially because of computer architecture or other reasons.



(a) A quadtree represented in a two-dimensional plane



(b) A quadtree represented in a tree structure

**FIGURE 2.2** Quadtree visuallization

Gershon Kedem descirble a new data structure *Quad-CIF* based on the quadtree specif-



ically for Integrated Circuit (IC) design in [30]. A typical quadtree structure begins with a large rectangle which covers all objects, and then divides itself to four equal size sub-rectangles as shown in Figure 2.2(a). The division is recursively carried on the sub-rectangles until the partition target is met, which is usually the number of partitions. An object is put into one quadtree node if and only if it is inside that node. Moreover, this object should not appear in any children nodes of the node it is in. This brings two problems for the IC design:

- 1) Most rectangles in IC design are small, so most of them will be at the leaves of the quadtree. This feature makes the depth of the tree to  $O(\log(N))$ .
- 2) There are many small objects on the top nodes, which returns a bulk collection of objects to most inquiries.

*Quad-CIF* changes the principle of how an object be stored in a node. It is a bottom-up approach:

- 1) Each collection of items is treated as a quadtree, with a minimum bounding box as the root;
- 2) If the item is a rectangle (object), it will be stored as it-is;
- 3) If the item is a collection of other items, its minimum bounding box will be stored with others represented by a pointer.

*Quad-CIF* has fewer levels than a regular quadtree when most of the dataset are small objects. In *Quad-CIF*, each cell is only checked one time and only interactive cells are checked. Since many cells are not interactive in the IC design, it reduces the time needed for sorting and querying by using *Quad-CIF*.

Hanan Samet in [31] and Allen Klinger in [32] introduce the concept of *region quadtree* in [31]. A quadtree is a *region quadtree* if it meets the following criteria:

- 1) its maximal blocks are disjoint with each other;
- 2) it has a standard size, i.e., powers of two;
- 3) it is at standard locations.

We can get a region quadtree by successive subdivision of an image array into four equal-size quadrants. The array is continuously divided until all elements inside it are entirely 0s or 1s. The region quadtree can be characterized as the resolution of that image since every node of it only has same elements.

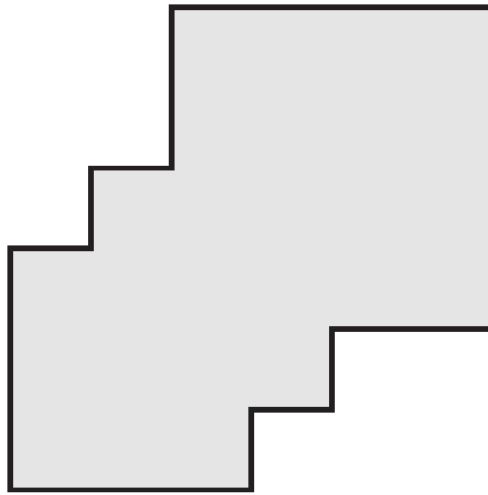
2.3 gives an example of the region quadtree. A simple polygon is shown in Figure 2.3(a) [31] and represented in a  $9 * 9$  binary matrix in Figure 2.3(b). By continuously dividing the binary array until all elements inside it are entirely 0s or 1s, a region quadtree is built as in Figure 2.3(c). Noted that the number in the cells are identifiers of the quadtree node. Figure 2.3(d) is the abstract region quadtree.

In the tree representation, the root node covers the entire array. Each quadrant of a node becomes a child node (labeled by nw (northwest), ne (northeast), sw (southwest), se (southeast)). The leaf nodes, as being presented, can not be further divided since it only has one of 0s or 1s. A leaf node is black if its corresponding block is entirely inside the represented region (Figure 2.3(a)), or white if it is entirely outside. The non-leaf nodes are all in gray.

#### 2.2.4 Multi-Jagged Partitioning

Parallel partitioning in parallel is tricky. Assume the spatial data is distributed in the distributed memory and without any duplications. The skew of the spatial data leads to the load imbalance problem for the partitioner while the purpose of spatial data partitioning is to mitigate load imbalancing. Parallel partitioners like Recursive Coordinate Bisection (RCB) [33] migrates data during partitioning to mitigate the load imbalance problem. Similar to the quadtree approach, RCB recursively bisects a node into two parts by cutting the plane orthogonally. The cutting also happens on the longest dimension. Thus, the weights of objects of two sub-plane are equal. In each recursion the data migration happens and which may lead to intensive data movement during RCB partitioning.

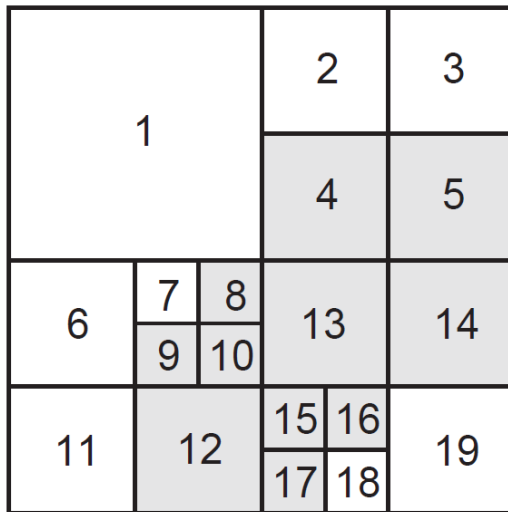
To address the data movement problem in RCB, Mehmet Deveci's team uses a parallel partitioning algorithm named Multi-Jagged coordinate partitioning (MJ) in [8].



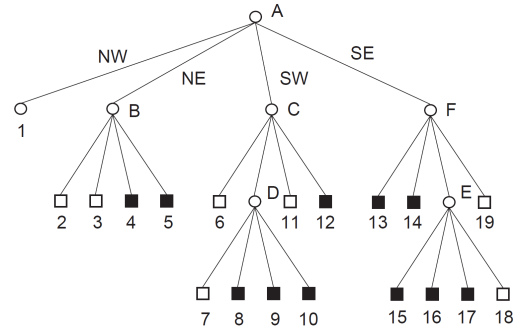
(a) Sample region

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	0	0
0	0	1	1	1	0	0	0

(b) The binary array representation



(c) The maximal blocks



(d) The corresponding quadtree

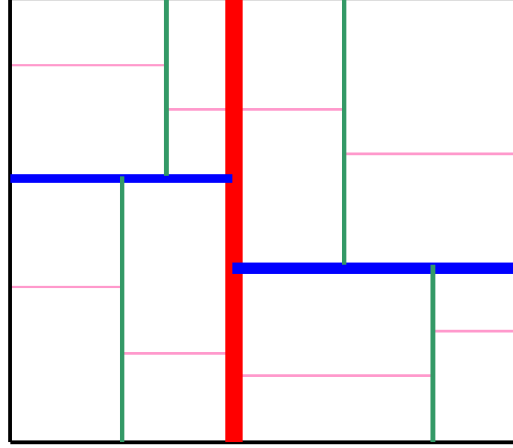
**FIGURE 2.3** An example of the region quadtree. (Originally presented by Hanan Samet).

There are differences between RCB and MJ:

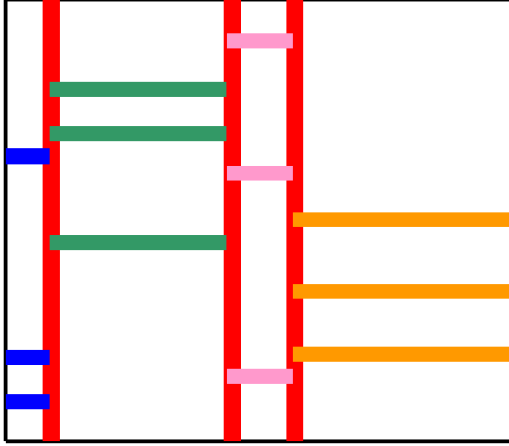
- 1) MJ cuts multi-partition concurrently, while RCB only cuts on one partition;
- 2) The data migration is optional in MJ, while RCB migrates objects after each bisection;
- 3) And MJ uses hybrid MPI + OpenMP for its implementation.

Figure 2.4(a) gives an example of using RCB to partition a two-dimensional area to 16 parts. The cutting order is sequential and indicated by the thickness of lines, beginning with the thickest red line. Two examples for partitioning the same area using MJ are given in Figure 2.4(b) and Figure 2.4(c). There are five recursions in Figure 2.4(b) and each

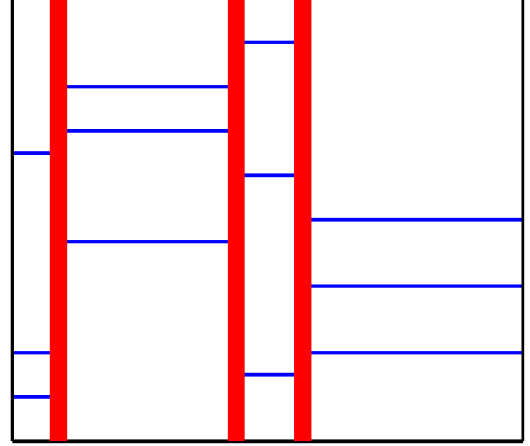
recursions three cutting operations happen concurrently, in which no data is moved. With data movement in MJ, as shown in Figure 2.4(c), the number of recursions is reduced to two, and the second recursion has twelve cutting operations (horizontal blue lines) happened concurrently.



(a) A partition of size 16 using RCB



(b) A partition of size 16 using MJ with no migration



(c) A partition of size 16 using MJ with migration

**FIGURE 2.4** Examples of using RCB and MJ. The cutting order is either indicated by colors (red  $\rightarrow$  blue  $\rightarrow$  green  $\rightarrow$  pink  $\rightarrow$  orange) or by the thickness of the cutting line (thickest to thinnest). The two MJ variants produce similar partitions, but MJ with migration allows more concurrency during partitioning. (Originally presented by Mehmet Deveci)

The overall view of MJ is described in Algorithm 2. MJ takes a spatial dataset  $C$  with  $n$  (d-dimensional) coordinates, and the target partition number  $\kappa$ . MJ maintains a permutation array of size  $n$  to avoid unnecessary data movement, marked as *Premute*. Another permutation array *xPerm* is used to record the beginning and ending indices of each

partition. Initially the number of partition  $P$  is 1. The **DETERMINE\_DIMENSION\_TO\_PARTITION** function returns the dimension that needs to be partitioned during this recursion. Note that in Algorithm 2, line 12 to line 17 can be done concurrently, and this is the concurrency we have seen in Figure 2.4.  $\mu$  records the balancing information of current partitioning, which is computed by function **PARTITION\_ON\_ONE\_DIMENSION** by comparing the sizes of newly generated partitions. **PARTITION\_ON\_ONE\_DIMENSION** also performs the partitioning operation and computes the weights of all partitions. After one concurrent recursion, **CHECK\_AND\_MIGRATE** checks if there is need to migrate any data. Finally, the information is recorded in  $xPerm$  and which will be returned as the final partitioning result.

---

**Algorithm 2** Parallel Multi-Jagged Algorithm

---

```

1: Input: Spatial data set  $C$ , its size  $n$ , weights of all objects  $W$ , dimensions  $d$ , target
   partition number  $\kappa$ .
2: Output: Partition information recorded in  $xPerm$ .
3: /* Initialize permutation */
4: for  $i$  from 0 to  $n - 1$  do
5:    $Permute\_i \leftarrow i$ 
6: end for
7:  $xPerm\_0 \leftarrow 0$ 
8:  $xPerm\_1 \leftarrow n$ 
9: Current number of partitions  $P \leftarrow 1$ 
10: while  $P < \kappa$  do
11:    $dim$  gets DETERMINE_DIMENSION_TO_PARTITION()
12:   for  $i$  from 0 to  $P - 1$  do
13:      $begin \leftarrow xPerm\_i$ 
14:      $end \leftarrow xPerm_{i+1}$ 
15:      $\mu \leftarrow \text{PARTITION\_ON\_ONE\_DIMENSION}((C\_i), W, Permute, begin, end,$ 
        $dim)$ 
16:      $newxPerm \leftarrow \text{UPDATE\_PERM}(Permute, begin, end, \mu)$ 
17:   end for
18:   CHECK\_AND\_MIGRATE(( $C$ ),  $W$ ,  $\mu$ ,  $\kappa$ ,  $Permute$ )
19:    $xPerm \leftarrow newxPerm$ 
20:   update  $P$ 
21: end while

```

---

Though the initial idea of MJ is to partition the original dataset into power of two par-

titions, there is a technique can be used to allow MJ to get arbitray number of partitons. When the dimension  $d$  and the target number of partitons  $\kappa$  are given, MJ can partition the data into  $\lceil \kappa^{1/d} \rceil$  initially, and then update the target number of partitons for current recursion based on the weights of partitons. With the dynamic target numbers of partitons, the eventual number of partitons will meet the initial requirement, which is  $\kappa$ .

### 2.2.5 Rectangle Tree

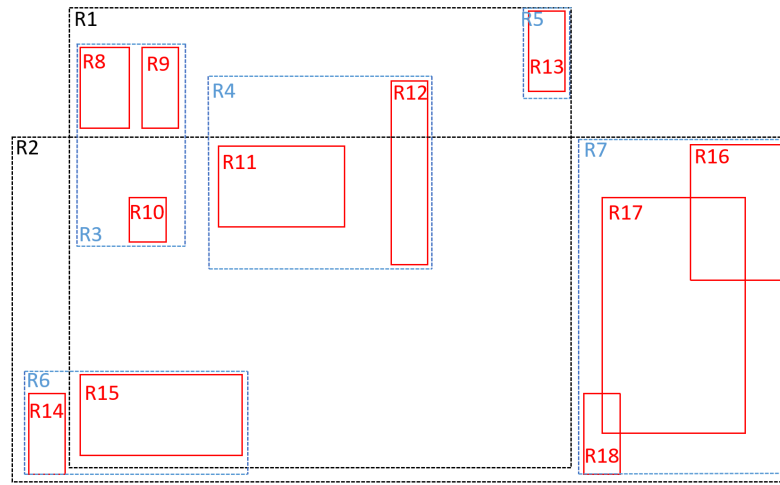
The Rectangle Tree (R-tree) is a popular data structure for indexing spatial data. The core idea of building R-tree is to group nearby objects in a node and several such nodes are grouped to form a node of the next higher level. A simple example of R-tree is shown in Figure 2.5(a). The objects in a child's node is always covered by its parent node. This feature allows quick searching in an R-tree by eliminating mismatch nodes and all of their children. Figure 2.5(b) is an abstract R-tree of the graph shown in Figure 2.5(a). The colors and nodes are corresponded in the two figures. The non-lefaf nodes contain the entries to their children and the leaf nodes contains the objects (point, rectangle, polygon, etc.) from the orginal spatial data set.

Though R-tree shows advantages on indexing or storing spatial data, its overlapping problem needs to be solved. For instance, the major parts of  $R1$  and  $R2$  are overlapped, which can indicate the usgae of space in the simple R-tree is inefficient. This problem leads to multiple chanllenges:

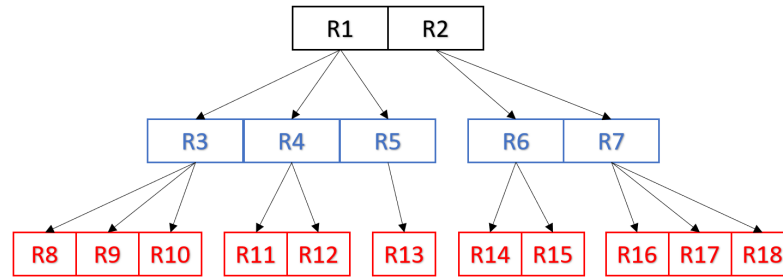
- 1) The area that a node covers can be minimized;
- 2) The overlap area of two nodes on the same level can be minimized;
- 3) The utilization of storage in R-tree can be optimized.

To address this overlapping problem, Norbert and Hans-Peter improved the performanc of Rtree in [34] and name their approach R\*-tree.

The first step in R\*-tree is to find the suitable subtrees. In order to minimize the overlap when inserting a new spatial object to the existiing tree, the new object should always be grouped with the least expansion of exisiting area in the tree. There are three scenarios:



(a) A rtree represented in a two-dimensional



(b) A rtree represented in a tree structure

**FIGURE 2.5** Rtree visuallization.

- 1) The current node is null; then a new node is built based on the new object;
- 2) The current node contains leaves, which indicates this node is next to the lowest level of this subtree; then the child node with the least area enlargement by grouping the new object is choosen.
- 3) The current node contains no leaves, the child node with the least rectangle enlargement by grouping the new object is choosen.

This step improves the utilization of R-tree space and improves the query performance as less area is needed to be searched.

When a node contains too many children, it is necessary to split this node to maintain the efficiency of an R-tree. The children are sorted by the values of their overlapping rectangles. Go through all the children and find a splitting position where the sum of the two sub-areas is smallest. Since the children nodes are sorted, this steps takes  $O(m \log(m))$  time, where

$m$  is the number of children in the node to be split.

R-tree is nondeterministic in inserting new spatial objects, which means different order of insertion can lead to different R-trees. Thus, in some cases, the R-tree can be inefficient because a high tree may be built. R\*-tree uses forced reinsert during its construction to improve the retrieval performance. During the step of finding a suitable subtree, there can be two choices based on the number of existing children

- 1) If the number of existing children is larger than a fixed limit, perform a split;
- 2) Otherwise, perform an insertion.

Notice that when a node is split, its parent node is also possible to meet the splitting criterion. R\*-tree forces a reinsertion when a split happens on a level for the first time. The reinsertion step is to remove several children of given node to reduce the size of the node. The removal should be performed in a way that the node covers minimal area after removal. The removed children then are reinserted into the R\*-tree. The advantages of the forced reinsertion are:

- 1) It changes the children of a node so that the node covers a smaller area;
- 2) The storage utilization is improved since less overlap between nodes on a same level;
- 3) Fewer splits happen because reinsertions take place;
- 4) Some nodes have chances to be placed in a better position of the R-tree, which improves the efficiency.

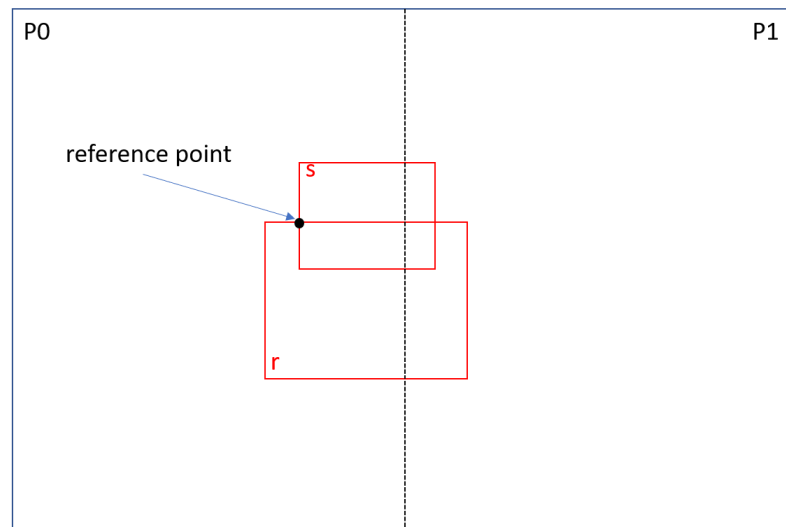
### 2.2.6 Duplication Avoidance

Duplicated results will be collected if replication happens (very likely) during partitioning. Considering the example in Figure 2.1, if a pair of overlapping objects is on a partition boundary, then this pair of objects will be stored in both partitions, which leads to the duplication problem when the partitioned data is used. To avoid misleading results and redundant computing, duplicate results are expected to be removed from the candidate set during the refinement stage.



A simple way to remove the duplications is to sort the candidate set generated in the filtering stage. It is easy to remove redundant candidates if all candidates are sorted. In some spatial join techniques, sorted candidates are required, and this simple duplication avoidance introduces no extra cost. However, in many cases, the entire candidates are too many to be put in the memory. In this scenario, an inline duplication avoidance technique is more suitable.

Jens-Peter Dittrich and Bernhard Seeger introduce an inline duplication avoidance technique in [35, 36] named *reference point method*. *Reference point method* finds a point within the intersection of MBRs of the two objects. An example is shown in Figure 2.6. Two rectangles  $r$  and  $s$ , belong to  $R$  and  $S$  respectively, lay on the board of two partitions  $P0$  and  $P1$ . Both partitions have  $r$  and  $s$ . Therefore, a spatial join approach will join  $r$  and  $s$  twice. Not only the results are redundant, but also the time cost are doubled. If some spatial objects (e.g. the Pacific Ocean on a typical Earth map) lay over multiple partitions, the cost of related computing may be enlarged exponentially, since larger objects are likely to have relations (cover, within, intersect) with more other objects.



**FIGURE 2.6** An example of using *reference point method*.

The idea of reference point method is to use a unique point within the intersection of  $R$  and  $S$ . The position of this unique point can be arbitrary, but the way to choose a reference point should be fixed in one program. In Figure 2.6, the top-left point of the intersection is

used as the reference point. As the reference point locates only in partition P0, the pair of  $R$  and  $S$  can be ignored when processing partition P1. This method adds few overhead to the entire program since the MBRs of objects can be used for generating reference points. The cost is  $O(1)$  for a single pair of candidates applying *reference point method*.

## 2.3 Spatial Join

### 2.3.1 Plane Sweep

*Plane Sweep* is introduced by Michael Shamos in his book and the most recent version is [37]. As one of the most basic algorithmic paradigms in computational geometry, the core idea of a plane sweep algorithm is using an imaginary, vertical or horizontal line to sweep the entire plane. The objects on the plane are only processed when the sweepline reaches them. Since if two geometries have common area, they will be reached by the sweepline in a certain window. Hence, many spatial data related operations (e.g. intersects) can be optimized to a one dimensional interval intersection problem by applying the plane sweep paradigm.

Lars Arge's team Introduced Scalable Sweeping-Based Spatial Join (SSSJ) in [11]. SSSJ performs an initial sort and classification on all objects, and then performs join operations using plane-sweeping. The first step is named *textitRectangle Join* in SSSJ. All objects in  $R$  and  $S$  are sorted by their MBRs' lower y-axis values into one list  $L$ . Then, the entire space is partitioned into  $k$  vertical stripes such that each strip contains at most  $2N/k$  objects, where  $N$  is the number of objects in  $R$  and  $S$ . An object is considered *small* if it is only in one stripe, otherwise it is *large*. A large object then is partitioned into three parts, two *end pieces* and one *center piece*. These pieces are used to avoid an object spanning on the sweepline for too long.

The next step is to perform plane-sweep based spatial join on sorted  $L$ . There are multiple choices for this step as long as they meet Algorithm 3. Algorithm 3 describes a general approach of a plane sweep join. The key data structure is used to store the elements being swept.  $D_R$  and  $D_S$  are two instances of such data structure. This data structure need to

support insert, scan and delete operations. The insert operation is used to insert new objects into  $D_R$  and  $D_S$ . The scan operation is used to find intersection between existing objects with newly inserted objects. The delete operation is used to remove objects that will not be future used in  $D_R$  and  $D_S$  to keep this data structure efficient. By finding the intersections between new inserted objects and previous objects in  $D_R$  or  $D_S$ , all intersected pairs can be found in one sweep. Algorithms like Tree Sweep [38], List Sweep, Striped Sweep, Forward Sweep [39] and so on, all can be used by SSSJ.

---

**Algorithm 3** General Plane Sweep Join

---

```

1: Input: Spatial data sets  $R$  and  $S$ .
2: Output: Join results  $RESULTS$ .
3:  $D_R \leftarrow$  empty
4:  $D_S \leftarrow$  empty
5: while  $R$  and  $S$  are not empty do
6:    $r \leftarrow R.head$ 
7:    $s \leftarrow S.head$ 
8:   if  $r.minY < s.minY$  then
9:     Insert  $r \rightarrow D_R$ 
10:    Delete elements in  $\rightarrow D_S$  whose  $maxY < r.minY$ 
11:    Find intersections between  $r$  and all element in  $D_S$ 
12:    Remove  $r$  from  $R$ 
13:   else
14:     Insert  $s \rightarrow D_S$ 
15:     Delete elements in  $\rightarrow D_R$  whose  $maxY < s.minY$ 
16:     Find intersections between  $s$  and all element in  $D_R$ 
17:     Remove  $s$  from  $S$ 
18:   end if
19: end while

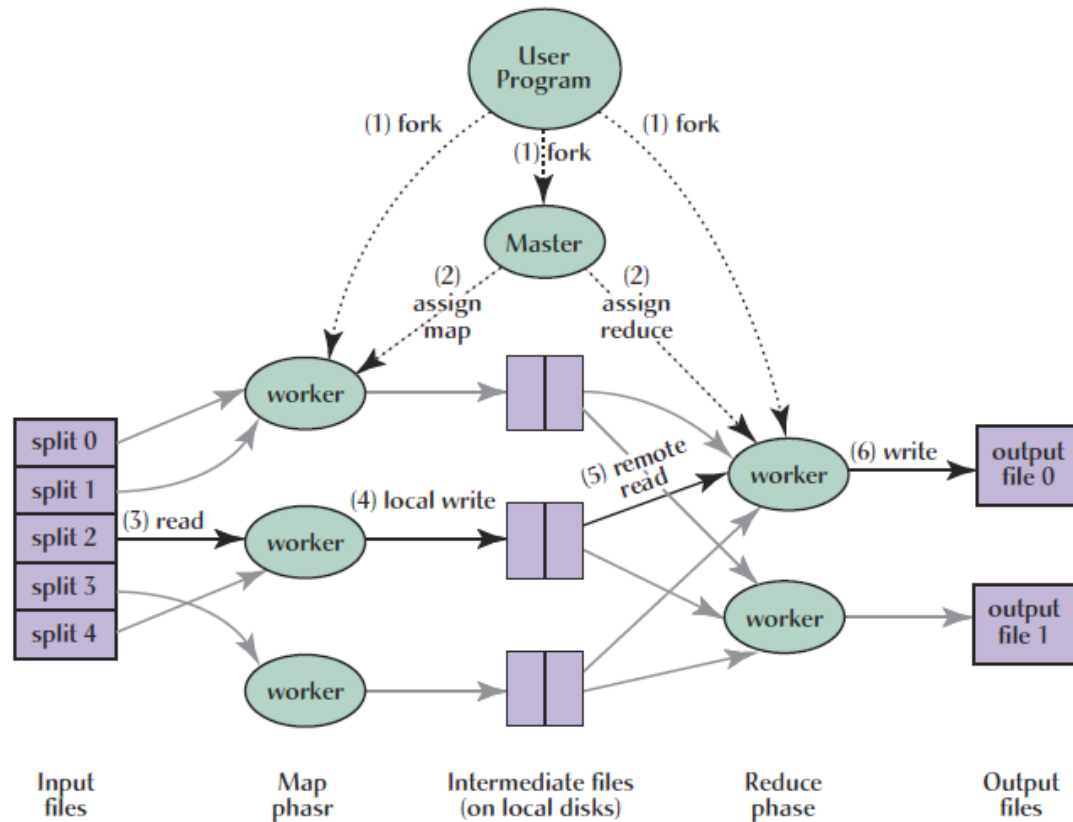
```

---

### 2.3.2 Spatial Join using Map-Reduce

*MapReduce* is a widely used parallel programming paradigm introduced in [40] by Jeffrey Dean and Sanjay Ghemawat. MapReduce is combined by two procedures: *map*, and *reduce*. In the map procedure, the system automatically schedules computing tasks among computing processes across large-scale clusters. Error handling and internode communications are also handled during this procedure. The "reduce" procedure is customized. The

programmers can decide how the mapped tasks are handled. MapReduce makes it is much easier to program on a distributed environment, provide load balancing options, and handle errors like a machine failure. A typical mapreduce framework is shown in Figure 2.7.



**FIGURE 2.7** An overview of the mapreduce paradigm. (Originally presented by Jeffrey Dean).

Based on mapreduce, Shubin Zhang's team presents Spatial Join with MapReduce (SJMR), a parallel spatial join algorithm to support heterogeneous relation queries for spatial datasets. The SJMR algorithm operates in the following two stages

- 1) **Map stage** Every object in  $R$  and  $S$  is presented by a unique identifier (OID), its MBR and other spatial properties if applicable. By using a tile-based partitioning method, every object is mapped into one or more partitions. Key-value pairs are generated where the keys are the partition numbers and values are OID, MBR and other spatial properties. SJMR treats one partition as one task in mapreduce.
- 2) **Reduce stage** The spatial join happens in this stage, including filtering and refinement steps. In the filtering step, a striped plane sweep (introduced in Section 2.3.1)

technique is used to find all candidate pairs. A list of OID pairs standing for candidates is generated in this step. The refinement step performs the required join operation (intersects, union, within, etc.) on the OID list. The final spatial join result is produced after this step.

SJMR uses *reference point method* (introduced in Section 2.2.6) as its duplication avoidance mechanism.

### 2.3.3 GeoSpark

*GeoSpark* is a large-scale spatial data processing framework running on in-memory clusters, presented by Jia Yu's team in [41]. Figure 2.8 gives an overview of GeoSpark.

GeoSpark has three layers:

- 1) **Apache Spark Layer** provides basic Spark functions such as loading or storing data on external memory, and regular RDD operations<sup>1</sup>;
- 2) **Spatial RDD Layer** extends the regular RDD to support spatial data;
- 3) **Spatial Query Processing Layer** provides spatial data querying functions such as building indices or performing regular spatial join functions such as intersects.

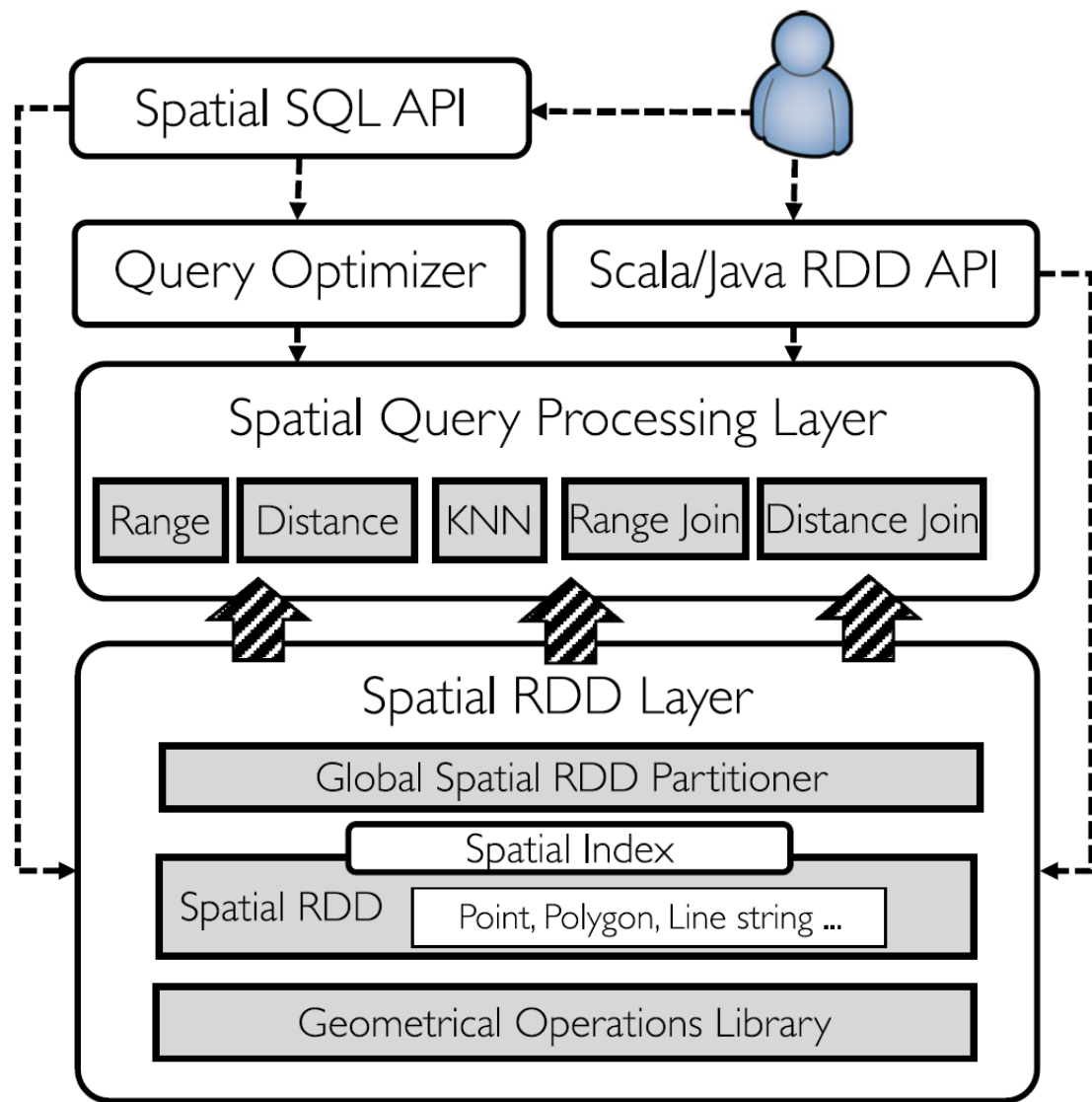
The Apache Spark Layer allows customized programming by using GeoSpark as a platform. Programmers can extend GeoSpark based on their own analyzing methods by modifying Spatial SQL API and Scala/Java RDD API.

Spatial RDD Layer is able to handle multiple formats of spatial data, such as Well-known Text (WKT), CSV, ESRI Shapefile and so on. GeoSpark also extends the default Spark data partitioner to handle spatial objects in this layer. Spatial indices, such as R-tree and quadtree, are also supported by GeoSpark.

Spatial Query Processing Layer handles spatial data querying. As spatial join is computing and data intensive, Spark makes massive data shuffling during querying, which is a burden to the network and also downgrade the performance. To mitigate this problem,

---

<sup>1</sup>Resilient distributed dataset (RDD) is a concept introduced in Apache Spark which stands for fault-tolerant data which can be read or modified in parallel.



**FIGURE 2.8** An overview of the GeoSpark framework. (Originally presented by Jia Yu).

GeoSpark partitions the spatial data based on the objects' locations. It also caches the Spatial RDD. Therefore, GEOSPARK join utilizes existing spatial partitioned RDDs and eliminates partitions which have no relations with current in processing RDDs. GeoSpark builds local indices on each RDD partition, instead of building a global index which enlarges the storage need. When a query is called, GeoSpark is able to partition it into small tasks and processed in parallel because of the distributed local indices.

## 2.4 Message Passing Interface

### 2.4.1 Non-Blocking Communication

The MPI blocking communications are easy to be understood and implemented. However, three problems are introduced:

- 1) Extra synchronization step is required;
- 2) The processes involved are busy waiting, i.e., they occupy all CPU circles during waiting;
- 3) It's hard to tell if some involved processes are working on something else or simply in failure.

By using non-blocking communication, not only the three problems are addressed, a program can overlap its computing tasks and communicating tasks.

MPI non-blocking APIs can be called by passing a communication, a request handle and other information similar to blocking APIs. A checking function (e.g. `MPI_Wait`, `MPI_Test`) needs to be called after a non-blocking communication API being used. The checking function checks if the purpose of the non-blocking has achieved. Whether the purpose of the non-blocking communication has been achieved or not, the checking function terminates itself. The main thread can resume its work and check the communication in future if the checking function returns a false answer. The program may even turn itself in a timed sleep and yield the CPU to other programs. Additionally, for the purpose of fault-tolerance, checking can be performed during the waiting phase of non-blocking communication. Figure 2.9 gives a simple example of using `MPI_Ibcast`. Instead of waiting the `MPI_Ibcast` to be finished, the process can perform other tasks, computing or even trigger another MPI communication.

Torsten Hoefer's team introduces two ways to extend MPI to support non-blocking collective communications in [42].

The first one is to use a separate thread which performs blocking collective communications to achieve the non-blocking purpose. A task queue model is used to manage these

```

MPI_Ibcast(&sendRecvBuffer, size, datatype, root, communicator, &request);
bool isReceived= false;
While (isReceived == false)
{
    .....
    /* computation, other MPI communications, etc. */
    .....
    MPI_Test(&request, & isReceived, MPI_STATUS_IGNORE);
}

```

**FIGURE 2.9** An example (broadcasting) of using MPI non-blocking APIs.

non-blocking communication tasks. One or more dedicated threads are initialized with their own task queues. Whenever a non-blocking collective function is called, a communication task (containing all data and necessary arguments) is push into one of the work queues, in a round-robin manner (if there are multiple communication threads). A worker thread can be notified either by checking its work queue, or using a condidion signal. A significant drawback is : it's recommended to assigne one MPI job one core, only half of the CPUs can be used to compute. Whenever a communication thread is notified there is a task in its work queue, this thread calls the corresponding MPI blocking collective operation. It set a done flag after the communication is finished. When the main thread calls a checking function, this done flag is checked. Another drawback of this method is that the communicators used by the communication threads should be duplicated from those communicators in the main thread. The duplication of an MPI communicator is a blocking collective operations. This drawback makes the first call to non-blocking collective operations on a communicator is actually blocking.

The second one is to use existing non-blocking point-to-point functions. The key idea of this way is to implement a collective schedule. This collective schedule needs to decide the oprations needed to accomplish the non-blocking collective operations. Similar to the blocking collective opeartions using blocking point-to-point opeartions, the non-blocking collective schedule also depends on non-blocking point-to-point opeartions. Those operations are saved in the collective shechdule instead of being checked immediately. A call of



communication checking function checks where those operations are fully accomplished or not.

#### 2.4.2 One-Sided Communication

MPI One-Sided Communication, also known as Remote Memory Access (RMA), is introduced by William Gropp and his team in [43, 44]. Classic MPI two-sided communication operations (blocking or non-blocking) bring in overheads such as extra memory copy, handshakes in Rendezvous protocol, operation match, and so on. By using one-sided communication, those overheads can be eliminated.

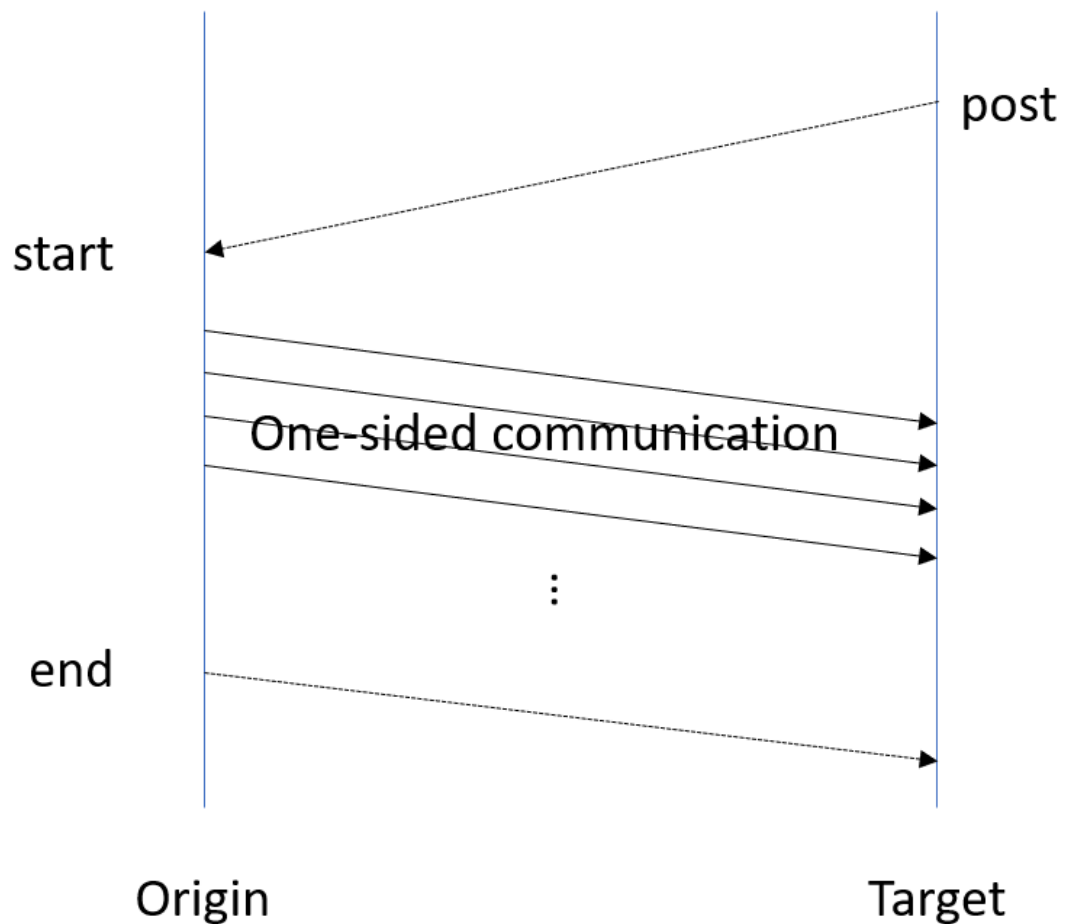
In MPI one-sided communication model, processes are allowed to access other processes' memory space directly. In two-sided communication, both sender and receiver need to provide complete parameters. As a contrast, only one process needs to specify the details in one-sided communication. As a benefit, the explicit parameters matching can be waived. MPI defines several terms for its one-sided communication model:

- 1) **Origin:** the process which performs the one-sided communication;
- 2) **Target:** the process whose memory is accessed;
- 3) **Window:** a memory area of the target process can be accessed by the origin process.

MPI one-sided communication does not mean that the origin process has access to the entire memory space of the target process; instead, only the window area is available. The time that the origin process is also bounded by the modes of one-sided communication. There are two modes in MPI one-sided communication:

- 1) **Active mode:** the target process decides in which time period its window can be accessed. The advantage is that the origin process(es) can perform multiple data transfers. An asynchronization (e.g. MPI\_Waitall) is required in this mode to finish all operations on the window.
- 2) **Passive mode:** the target process puts no limitation on when its window can be accessed. Instead, the target process need to execute the communication requests, either by setting a separated thread or periodically checking its window.

Figure 2.10 gives a simple example of active MPI one-sided communication. The communication begins with calling *MPI-Win\_start* and ends with calling *MPI-Win\_end*. These two functions specify the access period of the window on the target process. The target process posts its window by calling *MPI-Win\_start*. One-sided operations, such as *MPI\_Put*, *MPI\_Get*, *MPI\_Accumulate*, can be performed by the origin processes. *MPI\_Win\_fence* can be called during this period to perform a synchronization among all involved processes.



**FIGURE 2.10** An example of active MPI one-sided communication.

## CHAPTER 3

### EFFICIENT PARALLEL AND ADAPTIVE PARTITIONING FOR LOAD-BALANCING IN SPATIAL JOIN

#### 3.1 Introduction

Due to the developments of topographic techniques, clear satellite imagery, and various means for collecting information, geospatial datasets are growing in volume, complexity, and heterogeneity. For efficient execution of spatial computations and analytics on large spatial data sets, parallel processing is required. To exploit fine-grained parallel processing in large scale compute clusters, partitioning in a load-balanced way is necessary for skewed datasets. In this work, we focus on spatial join operation where the inputs are two layers of geospatial data. Our partitioning method for spatial join uses Adaptive Partitioning (ADP) technique, which is based on Quadtree partitioning. Unlike existing partitioning techniques, ADP partitions the spatial join workload instead of partitioning the individual datasets separately to provide better load-balancing. Based on our experimental evaluation, ADP partitions spatial data in a more balanced way than Quadtree partitioning and Uniform grid partitioning. ADP uses an output-sensitive duplication avoidance technique which minimizes duplication of geometries that are not part of spatial join output. In a distributed memory environment, this technique can reduce data communication and storage requirements compared to traditional methods.

To improve the performance of ADP, an MPI+Threads based parallelization is presented. With ParADP, a pair of real world datasets, one with 717 million polylines and another with 10 million polygons, is partitioned into 65,536 grid cells within 7 seconds. ParADP performs well with both good weak scaling up to 4,032 CPU cores and good strong scaling up to 4,032 CPU cores. We also applied ADP on GPU. Based on some of our experiments, performance benefit obtained by GPU-accelerated ADP is equivalent to that of 20 compute nodes (560 CPU cores).

---

The material in this chapter was presented in part on 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS). <https://ieeexplore.ieee.org/abstract/document/9139867>. Source code for this paper can be found at [https://github.com/Jayyee-HPC/paradp\\_partitioner.git](https://github.com/Jayyee-HPC/paradp_partitioner.git)

This chapter is organized as follows. Section 3.2 introduces background information and Section 3.3 introduces related work. Section 3.4 describes Adaptive Partitioning and its OpenMP based version. A parallel partitioning algorithm (ParADP) for ADP is presented in Section 3.6. Section 3.7 evaluates the performance of ADP and ParADP. Finally, Section 3.8 concludes this work.

## 3.2 Background

With the increasing volume and complexity of spatial data, there is an increasing demand for efficient geospatial techniques for parallelizing spatial computations [45]. Spatial join and map overlay are important in many scenarios like disaster prediction and rescue, urban planning and so on. Parallel processing can be used to speed up the compute-intensive and data-intensive spatial computations. Spatial data partitioning is an efficient method for data-parallel applications. However, spatial data is often skewed and contains a variety of geometric shapes, leading to a load-balancing problem in the parallelization of spatial computations.

Spatial join involves two spatial layers, namely,  $R$  and  $S$ . Performing spatial join queries with predicate *Intersects*, *Contains*, *Overlap*, etc., on  $R$  and  $S$  generates a collection of pairs  $(r, s)$ , where  $r \in R, s \in S$  that satisfy the join predicate. For example, "find all roads that cross a river" is an *Intersects* query [10]. A spatial join can be performed in two phases: 1) filter phase and 2) refinement phase. In the filter phase, the minimum bounding rectangles (MBR) of geometries are utilized to generate a collection of candidate pairs where each pair consists of cross-layer geometries whose MBRs have spatial overlap. These candidate pairs are further refined in the second phase using the actual geometric representations.

Many existing spatial data partitioning techniques are based on one layer, which ignore the distribution of data in another layer. In our prior work, we developed MPI-GIS for partition-based polygon overlay and spatial join computation [46]. MPI-Vector-IO is a component of MPI-GIS that performs parallel I/O of spatial data stored in parallel filesystems.

tem [29]. We experimented with uniform grid partitioning which does not perform well for skewed data. Adaptive spatial partitioning, as described in this paper, is designed to improve the load balancing in MPI-GIS.

1 2	2 20	1 10	2 1	1 20	2 20
3 6	4 9	3 3	4 5	3 18	4 45
Input A		Input B		Output C	

**FIGURE 3.1** Number of geometries shown in each grid cell. The workload of a cell in grid C is the product of the number of geometries present in corresponding cells in A and B (e.g., workload in the fourth cell is  $9 \times 5$ ).

To illustrate workload partitioning for spatial join, an example is provided in Figure 3.1 using A and B as the two input layers. A and B have different data distribution and the data is partitioned among four grid cells. The output is layer C which assumes the worst case scenario where a geometry in A needs to be compared against all geometries in B. In output grid C, the maximum workload is present in the fourth cell, even though the corresponding fourth cell in grid A and B do not have the maximum geometries in their respective grids. This is in contrast to the traditional partitioning algorithms which will prioritize dividing the second cell in grid A for example. However, neither partitioning on A nor B alone will focus on partitioning the actual workload. In this paper, we propose a Quadtree-based algorithm (ADP) based on both layers. ADP takes the distribution of geometries in both layers into consideration which can improve spatial partitioning by producing grid cells with similar workload.

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

Moreover, we propose a parallel adaptive partitioning algorithm (ParADP) in High Performance Computing (HPC) environment using MPI and C++ threads. Our distributed-

memory algorithm with  $p$  multicore processors scans the MBRs of an entire dataset by choosing  $p$  sample MBRs by each processor to get a global view of the data distribution. This is accomplished using the sample sort algorithm. For an HPC cluster with  $p$  multicore processors with  $q$  cores each,  $p$  vertical stripes and  $q$  horizontal stripes within a vertical stripe are created. The further partitioning is carried out by each CPU core in parallel to meet the user-defined number of partitions. This method is designed to keep the processors busy and minimize the overall data movement during spatial partitioning. Once the grid partitions are created and geometries are mapped to the grid cells, actual geometries can be finally moved to the corresponding cell(s) where they belong. Therefore, the actual geometries need to move only once from source processor to destination processor. This is in contrast to dynamic load-balancing approach where repartitioning is used after initial partitioning to distribute workload to processors with lighter workload [10].

Various experiments are designed to inspect the performance of ADP and Parallel Adaptive Partitioning (ParADP). As a sequential partitioning technique, ADP's and ParADP's partition qualities are compared with Quadtree partitioning and Uniform partitioning. Our implementations use Geometry Engine OpenSource<sup>0</sup> (GEOS) library which provides 1) spatial data indices such as Rtree, 2) geometry-based algorithms, and 3) parsing of spatial data.

The main contributions of this paper are as follows:

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

---

<sup>0</sup><http://trac.osgeo.org/geos>

GB) and 2) *parks* (9 GB) containing 75 million candidate pairs is completed within 7 seconds on a cluster of 4032 cores.

### 3.3 Related Work

Partitioning spatial data has been well-studied in literature. Equi-Partitioning, Min-Skew [47], Uniform grid, R-tree, Quadtree, and binary space partitioning are some classic examples of space partitioning. The choice of partitioning scheme depends on the application where it is used. Multijagged is a scalable spatial data partitioning algorithm [8]. However, it is applicable for point data only. In our lab's work, we consider polyline and polygon data as input.

**Parallel and Distributed partitioning:** Parallel data partitioning has been studied in the context of spatial query processing, spatial join operation, and polygon overlay [9, 29, 48–53]. SpatialHadoop supports different partitioning schemes using techniques based on Quadtree, R-tree, grid, etc. in a MapReduce environment [7]. PolySketch is a tile-based partitioning of polygons and polylines for GPU-based computations [54, 55]. The parallel partitioning algorithm presented here uses MPI.

**MPI-based GIS system:** In our lab's prior work, we experimented with MPI-based approaches [46, 56] and developed parallel I/O and partitioning framework called MPI-Vector-IO as an HPC system [29]. MPI-Vector-IO partitions WKT files stored in parallel filesystems like Lustre and GPFS into file splits. After data partitioning, a uniform grid is used for spatial partitioning. However, it suffers from load-imbalance for skewed data due to the lack of adaptive grid partitioning. This motivated the research into load-balancing spatial partitioning techniques.

**Load-balancing:** The output for an ideal spatial partitioning algorithm is to produce partitions that can be assigned to processors in a load-balanced fashion so that the total execution time is minimized. Ideally, processors should have equal amounts of work and a processor is not waiting for other processors to finish their computation. Both SpatialHadoop (SH) and Hadoop-GIS use dynamic load balancing present in Apache Hadoop

framework [7, 16, 57]. In [7], Quadtree method performed well relatively to other methods. To avoid the cost of reading and shuffle-exchange of the entire data, SH reads only a small percentage (e.g. 1%) of the data randomly to generate a global space partitioning. The methods presented here read all the MBR data. In my prior work, I studied load-balancing using Asynchronous Dynamic Load Balancing (ADLB) library, but the scalability was limited due to the high cost of moving polygonal data across MPI processes [58].

Data skew results in load imbalance. To mitigate the effects of load imbalance, SPINOJA system [9] partitions the spatial dataset such that the amount of computation demanded by each partition is equalized, and the processing skew is minimized. Heuristics like declustering skewed distribution of geometries and round-robin assignment of partitions to processors has been shown to be effective for loadbalancing [9, 59]. The experimental evaluation in [9] was done on a single processor with 8 cores. In this work, the performance has been evaluated using thousands of CPU cores in a distributed memory environment.

In this Chapter,  $ST\_intersection$ <sup>1</sup> and  $ST\_intersects$ <sup>2</sup> operations on two datasets are considered.  $ST\_intersection$  is used to find the intersection region of two geometries and  $ST\_intersects$  is used to find whether two geometries intersect with each other.

### 3.4 Adaptive Partitioning

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

The Adaptive Partitioning contains two steps: 1) find pairs of geometries from two spatial data layers whose MBRs overlap with each other, and 2) generate a grid using Quadtree

---

<sup>1</sup>[https://postgis.net/docs/ST\\_Intersection.html](https://postgis.net/docs/ST_Intersection.html)

<sup>2</sup>[https://postgis.net/docs/ST\\_Intersects.html](https://postgis.net/docs/ST_Intersects.html)



partitioning and map those pairs to the grid cells. In partition-based spatial join (PBSJ), for a given number of partitions (cells), geometries from each layer is stored in all the partitions where it belongs. Spatial join is then carried out in each partition. Instead of partitioning data from each layer, in this paper, we propose to first find all the candidate pairs, and then partition the candidate pairs on a grid. The advantages of this approach is workload-aware partitioning as well as reducing the inter-process communication.

### 3.4.1 Finding candidates for partitioning

---

#### Algorithm 4 Algorithm for finding candidates

---

```

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

```

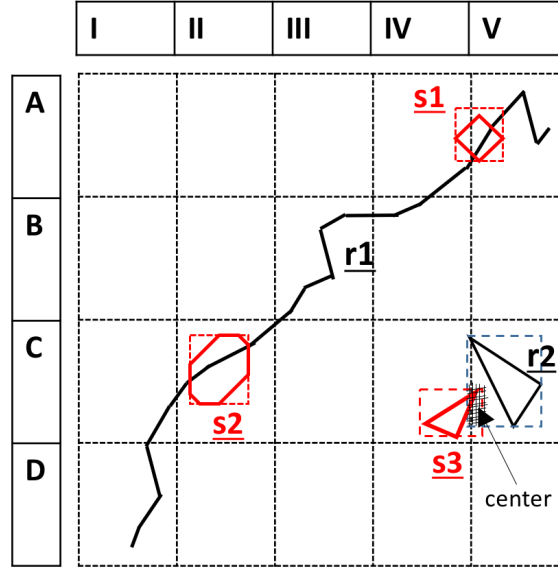
---

Algorithm 4 describes the procedure used to find the pairs of cross-layer geometries which potentially intersect each other, i.e. their MBRs have overlap. The inputs to the algorithm are two collections (layers) of geometries denoted by  $R$  and  $S$ . A Rtree index is built from MBRs of  $R$  which helps in reducing searching time [60]. Then an R-tree query is performed using MBRs of  $S$  which generates a collection of candidates denoted by  $C$ .

**Reference point method:** As described earlier, geometry spanning multiple cells of a grid is duplicated in all the cells it passes through.

As shown in Figure 3.2, a candidate  $(r_2, s_3)$  gets mapped to two cells (C, IV) and (C, V). To avoid redundant computation on the same candidate pair by two different processors in adjacent cells, reference point method is used [36]. As shown in Figure 3.2, this method

calculates the intersection of MBRs  $r_2$  and  $s_3$  of a candidate and assigns it to the processor owning the cell where the center point of the MBR intersection belongs. For  $(r_2, s_3)$ , the owner cell is (C, V). Since, this method works in the refinement phase, in a distributed memory implementation of PBSJ, mapping these geometries to their corresponding partitions requires data communication [10, 29, 57].



**FIGURE 3.2** Mapping of candidates to grid cells.  $(r_1, s_1)$ ,  $(r_1, s_2)$ ,  $(r_2, s_3)$  are candidates. Due to our output-sensitive method, geometry  $r_1$  is not stored in cell IDs (D, I), (C, I), (B, III), (B, IV), and (A, IV) even though it passes through these grid cells. Instead,  $r_1$  is stored in cells (C, II) and (A, V) because it is part of two candidates  $(r_1, s_1)$  and  $(r_1, s_2)$  only.

**Output-sensitive Duplication Avoidance technique:** We employ the reference point method in our implementation. This method can be applied to reduce redundant storage of geometries in spatial join. Our new method takes advantage of the fact that since all candidates are known, a geometry need not be stored in all the grid cells it passes through. The storage of geometries in grid cells can be determined by the location of the candidates in the grid. We illustrate this observation using geometry  $r_1$  that spans through multiple cells in Figure 3.2. The space-saving is one of the advantages of considering both layers during spatial partitioning. In a filter and refine based join processing, there can be thousands/millions of candidates. Less redundancy resulting from avoiding unnecessary storage of geometries leads to reduced storage requirements as the grid cells become finer in resolution. This also applies to minimizing the communication required to move the large geometries to their

corresponding cells in a distributed memory version of PBSJ.

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

### 3.4.2 Multithreaded Partitioning of Candidates

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

To illustrate an issue with parallelization of our Quadtree partitioning approach, here is an example. Let us consider 4 cells with weights given by an array  $A = \{20, 15, 6, 4\}$ . Let us assume that the first cell ( $A[0]$ ) got divided into four sub-cells with weights  $\{19, 1, 0, 0\}$  by a thread. If another thread picks cell with weight 6 for division instead of picking cell with weight 19, and we need only eight cells as output, it is clear that we may not end up with desired output. A single-threaded execution would pick 19 before 6 because of its

descending order priority. As, we can see, we do not want a lower weight cell to be considered for subdivision by a thread, if there are relatively higher weight cells still undergoing division by another thread. This decision-making also depends on how many cells have already been partitioned w.r.t the target  $N$ .

Theorem 3.1 is used for guiding the parallelism of Quadtree partitioning in ADP and ParADP.

**THEOREM 3.1** *Assume that  $A$  is an array of cells arranged in descending order of its cell-weights. A sequential algorithm partitions cells in  $A$  by selecting a cell with the largest weight, and splitting the cell into sub-cells by dividing its weight. If  $w_i \geq w_0/\kappa$ , a sub-set  $B = w_0, w_1, \dots, w_{i-2}, w_{i-1}$  can be partitioned into at most  $i * \kappa$  sub-cells whose values are  $\geq w_i$ , where split factor  $\kappa \geq 1$ .*

**Proof of Theorem 3.1:** If  $(i * \kappa) + q$  sub-cells were generated by the partitioning algorithm, whose weights  $\geq w_i$ , where  $q > 0$ , then  $\sum_{n=0}^{i-1} w_n \geq ((i * \kappa) + q) * w_i \geq i * w_0 + q * w_i$ . This leads to contradiction since  $\sum_{n=0}^{i-1} w_n$  can at most be  $i * w_0$ . ■

Based on Theorem 3.1, we can simultaneously split cells with weights  $w_0$  to  $w_i$  when  $w_i \geq w_0/\kappa$ . An algorithm can control the degree of concurrency by choosing  $\kappa$ . Moreover, by adjusting the value of  $\kappa$ , we can trade off speed-up vs accuracy of a parallel partitioning method. Here the accuracy means the similarity of the partitioning grid produced by parallel method compared to the sequential method.

By applying Theorem 3.1, we have incorporated a heuristic in our OpenMP algorithm which compares the weight of a cell against the cell with the maximum weight  $w_{max}$ . A cell other than the cell with  $w_{max}$  can be partitioned if its weight is  $\geq w_{max}/\kappa$ , where  $\kappa \geq 1$ . The value of  $\kappa$  can be customized. With lower value of  $\kappa$ , the output of parallel partitioning is closer to a sequential partitioning.

Algorithm 5 describes the OpenMP based Parallel Quadtree partitioning algorithm. A user provides the desired number of partitions in the grid.  $C$  is the set of candidate pairs. Elements in  $C$  contains points with weights. The weight of a cell is the summation of the weights of candidates in that cell. Initially,  $G$  only contains an MBR, which is denoted by

---

**Algorithm 5** OpenMP based Quadtree Partitioning Algorithm

---

```

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

```

---

*GlobalMBR* in the algorithm, that covers all objects in  $R$  and  $S$ . During the execution of the algorithm,  $G$  will contain sub-cells generated at a given time and elements of  $G$  are sorted by their weights in descending order. Cells whose weights are  $\geq \kappa * G[0].weight$  are subdivided concurrently via OpenMP tasks. The computations in Step 9 include distribution of the candidates in a cell among its sub-cells and calculating the weights of the new sub-cells.

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

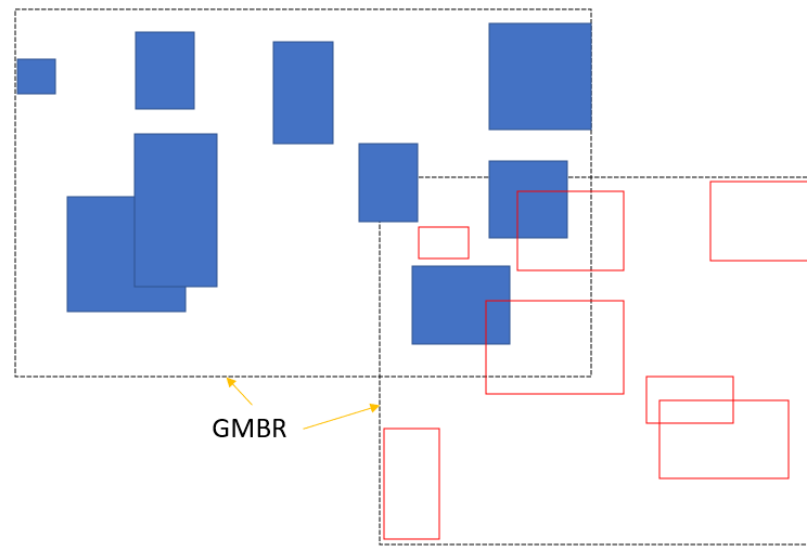
### 3.5 GPU Acceleration of Adaptive Partitioning

Given two input layers (files), we need to find out the candidates in order to estimate workload in spatial join. However, sequential enumeration of candidates based on Algorithm 4 on a CPU becomes time-consuming for large data sets. Algorithm 4 takes  $O((m + k) * \log(n))$  time using R-tree data structure, where  $n$  and  $m$  are the number of geometries in the two input datasets.  $k$  is the average size of Rtree query results. However, on a GPU, there is a lack of a fast algorithm with  $O((m + k) * \log(n))$  time.

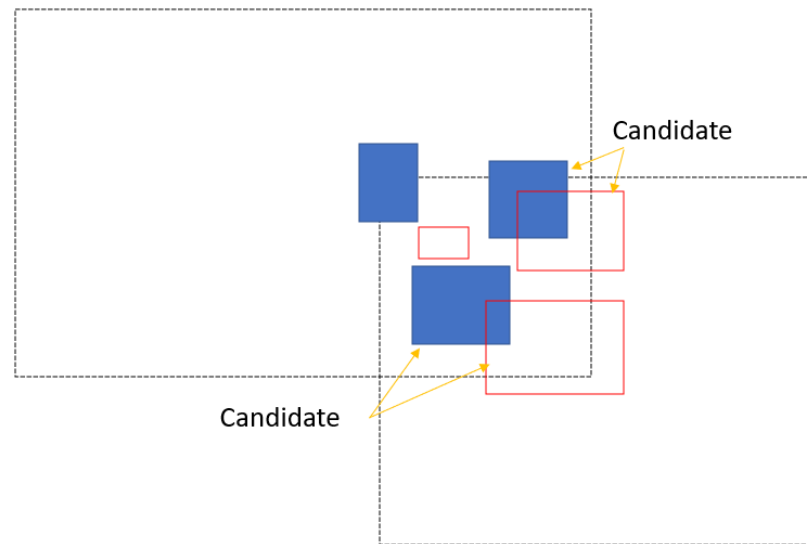
A brute force approach for finding candidates employs all-to-all intersection test on two input datasets. This can be easily implemented on a GPU. However, the performance will degrade for larger inputs because of the quadratic nature of the algorithm. For datasets with sparse output (small  $k$ ), this algorithm is inefficient and does not utilize the GPU parallelism effectively. Sparse output happens when MBRs from one dataset only intersects with a small set of MBRs from the other dataset.

We employ data partitioning as a divide and conquer strategy. Data partitioning of input MBRs leads to space partitioning. This leads to a filter-and-refine approach for finding

candidates [54]. Figure 3.3(a) shows the data partitioning approach where MBR approximations of geometries are shown as blue and red rectangles. The main idea is to cluster the MBRs of each layer after sorting of MBRs on x-coordinate. The clustering produces data partitions that we refer to grouped-MBRs or GMBRs. Next, in Figure 3.3(b) we show the filter step, where some MBRs that are not in the common area between the two GMBRs, are discarded from further processing. Finally, after reducing search space, candidate pairs are computed by overlap tests among the remaining MBRs.



(a) Two *GMBRs*



(b) Candidate pairs

**FIGURE 3.3** Visualization of Algorithm 6 and 7

To better understand GPU-ADP, we visualized the processes of Algorithm 6 and 7 as in Figure 3.3.

Figure 3.3(a) is the visualization of Algorithm 6. Two *GMBRs* are formed based on two sub-sets of MBRs.

Figure 3.3(b) is the visualization of Algorithm 7. Only MBRs within or intersect with the intersection of two *GMBRs* are considered.

GPU has been used on partitioning geospatial join using bitmap quadtree approach in [61], and building quadtree in [62]. Generally, quadtree approaches on GPU work on points without weight, while ADP uses MBRs with weights to partition. We need to solve these two problems to take advantages from GPU.

Algorithm 6 describes the implementation of a filter. The core idea of Algorithm 6 is to partition input MBRs into many groups of MBRs, and MBRs are geographically closed to each other in a subgroup. The input is a set of MBRs from one dataset. The output *Groups* is a set of MBR groups, along with the large MBRs of every group. The sorting procedure guarantees that MBRs in one group are geographically nearby. Sorted MBRs are divided into small sets of a given size, *GROUP\_SIZE*, to take advantage of the GPU architecture. A GPU block can be assigned one or more groups of MBRs. The block finds the overall MBR of a group, then push the result into *Groups*.

---

**Algorithm 6** Algorithm for implementing an MBR filter on GPU

---

- 1: **Input:** A collection of MBRs, *MBRs*.
  - 2: **Output:** A set of MBR groups denoted by *Groups*
  - 3: Sort(*MBRs*, by min x)
  - 4: Divide *MBRs* into subsets *SUBs* by size *GROUP\_SIZE*
  - 5: Assign *SUBs* to different GPU blocks
  - 6: **for** Each GPU block **do**
  - 7:     **for**  $sub_i$  in *SUBs* assigned to the block **do**
  - 8:          $GMBR_i \leftarrow$  compute the overall MBR of  $sub_i$
  - 9:          $Groups \leftarrow (sub_i, GMBR_i)$
  - 10:     **end for**
  - 11: **end for**
- 

After applying Algorithm 6 on both datasets, we use the outputs to find candidates on



GPU as shown in Algorithm 7.  $R\_Groups$  and  $S\_Groups$  are outputs of Algorithm 6. The output  $C$  is a collection of candidates with weights.  $rg_i$  and  $sg_j$  are groups from  $R\_Groups$  and  $S\_Groups$  respectively. Each GPU block is assigned one or more groups from  $R\_Groups$ , and find intersection for these groups with every group in  $S\_Groups$ .  $W_g$  is the total weight of all candidate pairs.  $W_g$  is used to decide minimum weight of a partitioned cell by dividing the target number of partitions.

---

**Algorithm 7** Algorithm for finding candidates on GPU

---

```

1: Input: Two sets of MBR groups denoted by  $R\_Groups$  and  $S\_Groups$ 
2: Output: Candidate set denoted by  $C$ , global weight  $W_g$ 
3: Assign  $R\_Groups$  to different GPU blocks
4: for Each GPU block do
5:   for  $rg_i$  in  $R\_Groups$  assigned to the block do
6:     for  $sg_j$  in  $S\_Groups$  do
7:       Find all intersections of all mbrs in  $rg_i$  and  $sg_j$ 
8:       Calculate all corresponding center points of intersection
9:       Calculate all corresponding weights using weight equation.
10:       $C \leftarrow C \cup tuple(intersections, centerpoints, weights)$ 
11:       $W_g \leftarrow W_g + sum(weights)$ 
12:     end for
13:   end for
14: end for

```

---

Algorithm 8 describes the GPU based Parallel Quadtree partitioning algorithm with CUDA dynamic parallelism. It is modified based on a sample code<sup>3</sup> provided by NVIDIA to work on candidates with weights to produce partitioning cells, while the original code only builds a quadtree index on points without weights. Due to the volume of the code, Algorithm 8 only provides the general idea. The detailed pseudocode can be found in A.1.

In Algorithm 8, only one block is called initially. Four children blocks will be launched if none of the stop criteria are met. The stop criteria include the customized number of target partitions  $N$ ,  $max\_depth$ ,  $minimum\_weight$ , and  $minimum\_candidates$ .  $max\_depth$  is the max depth of the quadtree approach can be, which is a GPU hardware limit. The limit of  $N$  can be lifted to get better partition quality but less controllable on the number

---

<sup>3</sup><https://github.com/NVIDIA/cuda-samples/tree/master/Samples/cdpQuadtree>

of produced cells.  $minimum\_weight$  can be calculated by  $W_g \div N \times \epsilon$ , where  $W_g$  is given by Algorithm 7 and  $\epsilon$  ranges from 1 to 3 to avoid over partitioning on light cells.  $minimum\_candidates$  is a fixed value to avoid a cell contains too few candidate pairs. When one of the stop criteria is met, the block writes its  $node\_MBR$  to  $G$  and exists.

Initially, the entire collection of candidate pairs  $C$  with the  $Global\_MBR$  as the  $node\_MBR$  are given to a single block. An  $node\_MBR$  strictly covers all candidates in its  $range$ .  $range$  is a pair of integers recording the start and end locations of  $C$ , initialed as  $(0, C.size)$ . The initial depth  $depth$  is set to 1.  $N$  is the customized number of target partitions.  $C$  is the set of candidate pairs with weights. The weight of a cell  $W_{cell}$  is the summation of the weights of candidates in that cell.  $GlobalMBR$  is divided into four equal-size sub-MBRs by computing its center point  $center$ .

---

**Algorithm 8** GPU based Quadtree Partitioning Algorithm

---

- 1: **Recursive function:** \_\_kernel\_\_ quadtree\_partition()
  - 2: **Input:** Candidate collection  $C$ , target number of cells  $N$ ,  $max\_depth$ ,  $minimum\_weight$ ,  $minimum\_candidates$ ,  $node\_MBR$ ,  $range$ ,  $depth$
  - 3: **Output:** A global list of grid cells  $G$
  - 4:  $W_{cell} \leftarrow$  sum weights of all candidates in  $range$
  - 5: **if** one or more stop criteria are met **then**
  - 6:      $G \leftarrow Node\_MBR$
  - 7:     **return**
  - 8: **end if**
  - 9:  $center \leftarrow node\_MBR$
  - 10: **Divide**  $node\_MBR \rightarrow$  four  $new\_MBRs$  based on  $center$  for four children
  - 11: Within  $range$ , **divide**  $C$  evenly and assign each thread one subset  $C_{sub}$
  - 12: **Count** the numbers of candidates in four  $new\_MBRs$
  - 13: **Scan** the threads' results to get the numbers of candidates in four  $new\_MBRs$
  - 14: **Divide**  $range \rightarrow$  four  $new\_ranges$  based on the scan results
  - 15: **Move** candidates to four sections based on the scan result
  - 16:  $depth += 1$
  - 17: **Launch** four quadtree\_partition() using  $new\_ranges$ ,  $new\_MBRs$ ,  $depth$  and other old parameters
- 

Each thread in a GPU block is assigned parts of the candidates  $C_{sub}$  within the  $range$ . A thread will first count how many candidates belong to each quarter. Thread 0 scans all counting results to make rooms for moving  $candidates$  from  $in\_candidates$  to

*out\_candidates*. Based on the scanning results, the input *range* is divided into four new *subranges* to record the number of candidates belong to each sub MBR. Each thread moves all candidates in its  $C_{sub}$  to their correct position in *out\_candidates*. *depth* plus one is used for the children blocks. In the end, one thread of this grid launches four new blocks using the updated parameters.

### 3.6 Parallel Adaptive Partitioning

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

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

#### 3.6.1 Parallel ADP for Distributed Memory

To speed up the partitioning algorithms, we designed a parallel partitioning algorithm, called ParADP, by using a hybrid MPI and multithreaded implementation. MPI is only used to facilitate data communication among the computing nodes. C++ Threads are used within each multicore node. ParADP consists of a *parallel MBR sorting phase*, a *data communication phase*, a *work distribution phase*, and a *partition phase*. Parallel Sorting by Regular Sampling (PSRS) technique [63] is used for sorting regular samples of MBRs taken from different compute nodes.

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

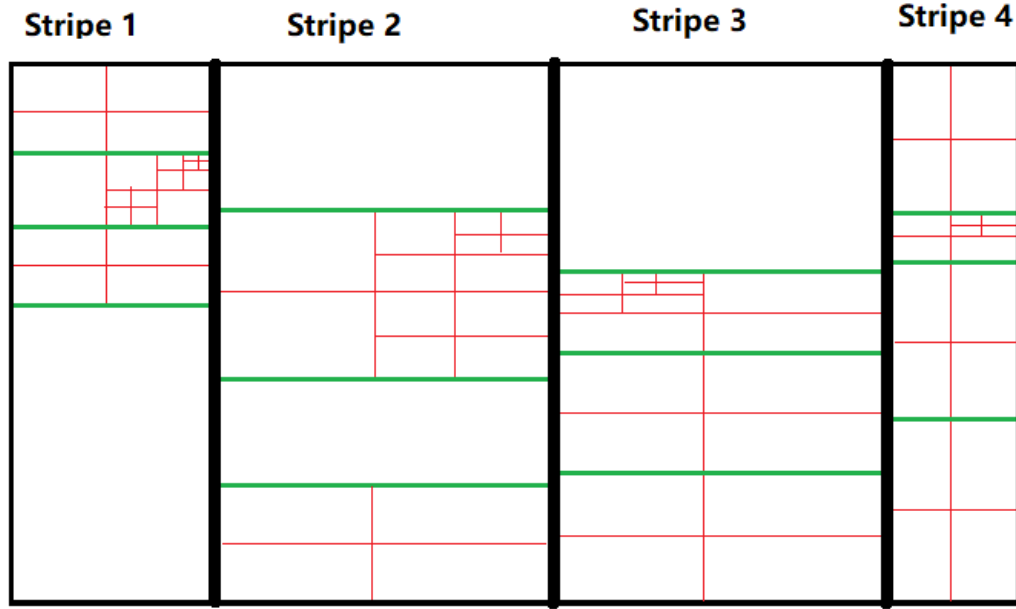
- 1) **Parallel Sorting Phase:** Each compute node reads  $m/p$  and  $n/p$  MBRs from the two datasets respectively. Then each node sorts the MBRs from **dataset I** by the maximum x-coordinate values. Each node chooses  $p$  regular samples and node 0

gathers all samples. Node 0 sorts all samples and chooses  $p - 1$  pivot values from the sorted sample list. Node 0 broadcasts all pivot values.

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

The grid generated by the parallel partitioning system is different from a normal Quadtree grid. An example is given in Figure 3.4 to show how ParADP works. Each processor in ParADP gets a unique stripe and divides the stripe further among its cores. To reach the user-defined target number of partitions, each core partitions the space within its horizontal stripe independently.

Figure 3.5 shows a grid with 8192 cells generated by ParADP for the *roads* and the *parks*



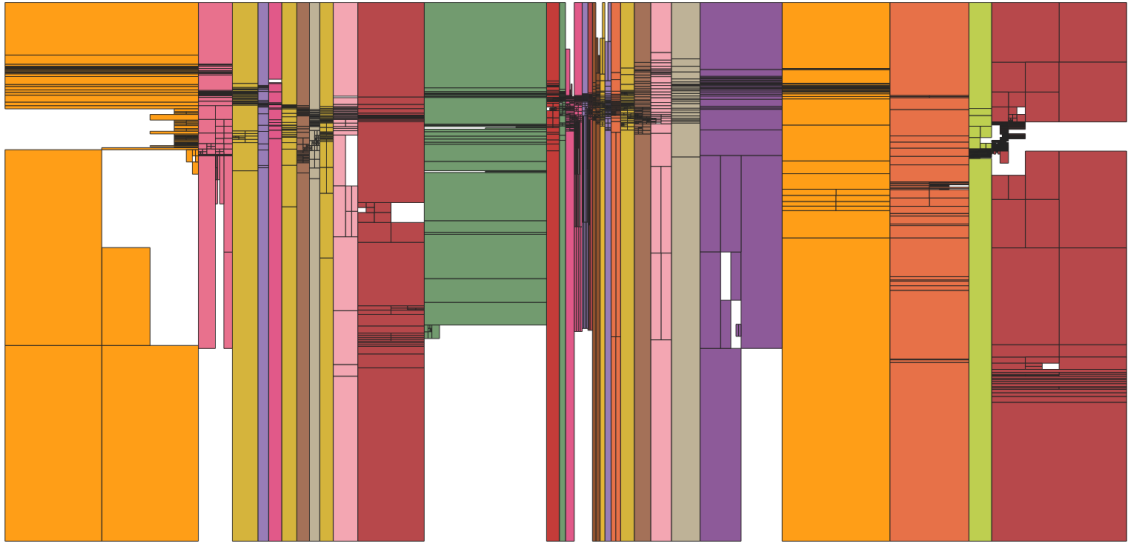
**FIGURE 3.4** ParADP using 4 compute nodes with 4 cores in each node. Longitudinal thick black lines are generated first for rearranging data based on its stripe boundary in each node. The green lines are generated by each node independently. Every CPU core/thread is assigned one cell. The thin red lines are partitioning boundaries generated by each CPU core individually.

using 32 nodes on *Bridges*. There are 32 stripes which can be distinguished by different coloring scheme.

### 3.6.2 Time Complexity

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

**Best and worst case:** ADP takes  $O(mn)$  time because an MBR in dataset I can potentially overlap with all the MBRs in dataset II. However, in ParADP, MBRs from dataset I



**FIGURE 3.5** Parallel partitioning of the *roads* and the *parks* into 8192 grid cells using ParADP

is roughly equally divided among  $p$  nodes. PSRS algorithm ensures that a processor ends up with at most  $2m/p$  objects [63]. If we assume that MBRs are drawn from uniform distribution, each compute node roughly gets  $m/p$  MBRs.

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

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

### 3.7 Experimental Results

All of our experiments used various real world data sets, *sports*, *lakes*, *parks*, and *roads*, which are taken from SpatialHadoop website<sup>4</sup>. The attributes of the datasets are shown in Table 3.1. ADP in this section refers to the sequential version.

Most of the experiments on CPU are done on a supercomputer named *Bridges*<sup>5</sup> at the

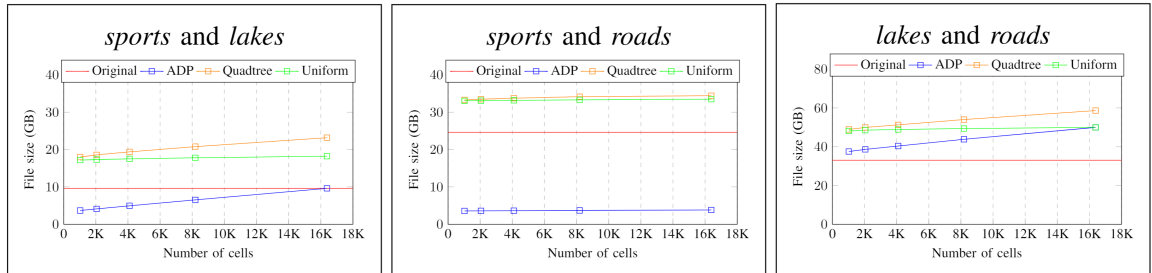
Name	Type	#Geometries	File size
<i>sports</i>	Polygons	1.8 M	590 MB
<i>lakes</i>	Polygons	8.4 M	9 GB
<i>parks</i>	Polygons	10 M	9.3 GB
<i>Roads</i>	Polylines	72 M	24 GB

**TABLE 3.1** Attributes of the data sets

Pittsburgh Supercomputing Center. *Bridges* has 752 regular nodes and each node has 2 Intel Haswell (E5-2695 v3, 14 cores each processor) processors running at 2.3 - 3.3 GHz with 128 GBs of memory. The experiments on GPU are done on a single Nvidia Titan V GPU. Titan V has Volta architecture. It has 12 GB HBM2 memory and 5120 CUDA cores.

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

### 3.7.1 Performance of Output-sensitive Duplication Avoidance



**FIGURE 3.6** Storage space needed using different partitioning techniques

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

<sup>5</sup>bridges.psc.edu

Then the geometries in each grid cell were stored separately in a file and written to hard disk in *WKT* format.

To evaluate the performance of our new duplication avoidance technique on the pre-processing stage of spatial join, ADP, Quadtree, and Uniform partitioning were used on different pairs of datasets. Their outputs were stored in a hard disk. We applied output-sensitive duplication avoidance technique in ADP only to compare the improvement in space complexity. In Figure 3.6, in all situations, ADP generates fewer data than Quadtree and Uniform partitioning. In the case of partitioning *sports* and *lakes* into 1024 cells, the total size of files generated by ADP only use around 10% of the disk space used by data generated by Quadtree and Uniform partitioning.

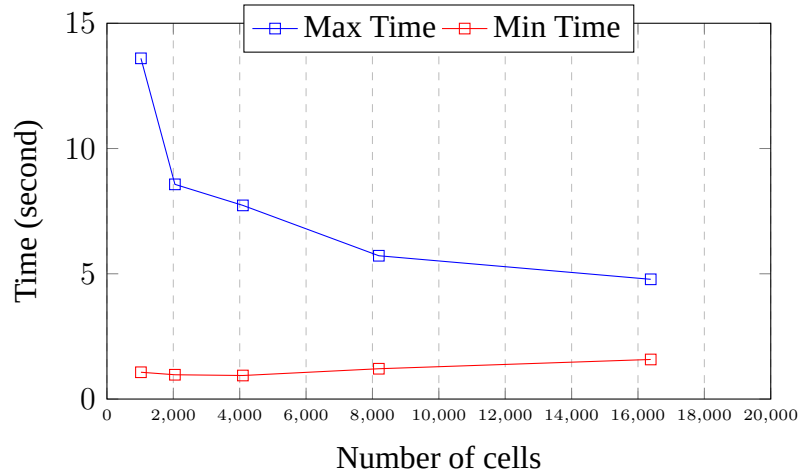
When one dataset is much smaller than the other one, the number of candidate pairs may be smaller than the number of geometries in the two datasets. ADP can take this advantage and save disk space when the partitions are written to disk. This also means less communication for an in-memory distributed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

### 3.7.2 OpenMP Quadtree Partitioning Speedup

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

As shown in Table 3.2, the benefits of OpenMP parallelization is realized for higher





**FIGURE 3.7** Max process time and min process time for GEOS *Intersects* method using *parks* and *sports* using an increasing number of grid cells generated by ADP. 84 cores are used for all cases.

number of cells. For 32K cells, OpenMP based Quadtree partitioning achieved its highest speedup of 2.63 compared to the sequential Quadtree partitioning. Performance is impacted by the lack of scalable multithreaded R-tree library that we use internally in Line 9 of Algorithm 5.

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

**TABLE 3.2** OpenMP based Quadtree Partitioning time (seconds)

### 3.7.3 Computing cost for ADP

We designed several experiments to test the impact on ADP quality using different partition numbers. *Parks* and *sports* data are partitioned into 1024, 2048, 4096, 8192, 16384 cells for five experiments respectively. For all cases, three nodes (84 cores) on *Bridges* are used. As shown in Figure 3.7, with a higher number of partitions, the gap between maximum and minimum MPI process execution times narrows for GEOS *Intersects* method. This demonstrates that load-balancing improves for a higher number of partitions for ADP.

Partitioning cost is determined not only by the number of objects in the two layers but also by the number of candidate pairs found during the filtering phase. The number of candidates found in the filtering phase using Algorithm 4 are as follows:

- 1) *Parks* and *sports* is about 2.7 millions.
- 2) *Roads* and *parks* is about 8.1 millions.

$R \oplus S$	Partition Number	$T_{cand}(s)$	$T_{quad}(s)$	$T_{total}$
<i>parks</i> $\oplus$ <i>sports</i>	8192	835.69	28.43	864.12
<i>parks</i> $\oplus$ <i>sports</i>	16384	828.93	43.10	872.03
<i>parks</i> $\oplus$ <i>sports</i>	32768	853.57	99.32	952.89
<i>parks</i> $\oplus$ <i>sports</i>	65536	859.63	346.73	1206.36
<i>roads</i> $\oplus$ <i>parks</i>	8192	19714.03	1222.69	20,936.72
<i>roads</i> $\oplus$ <i>parks</i>	16384	19193.30	1562.82	20,756.12
<i>roads</i> $\oplus$ <i>parks</i>	32768	19972.41	1975.74	21,948.15
<i>roads</i> $\oplus$ <i>parks</i>	65536	19829.00	2757.72	20,756.12

**TABLE 3.3** ADP total and break-down execution time for different pairs of datasets

Table 3.3 shows the time it takes for ADP on different pairs of datasets to generate a given number of partitions.  $T_{cand}$  is the time used for finding the candidates as in Algorithm 4. For the same pair of datasets,  $T_{cand}$  doesn't change as the number of candidates doesn't change much when the partition number changes.  $T_{quad}$  is the time for implementing Quadtree partitioning.

When the size of the two datasets increases, the time for finding candidates pairs grows faster than the Quadtree partitioning time. This is shown in Table 3.3. Partition based spatial join is affected by data partitioning [10] cost. From Table 3.3, we can see that partitioning can take a lot of time when the target partition number is large for bigger datasets. Even though ADP is effective as we saw earlier, it is time-consuming. This motivates the need for parallel partitioning.

$R \oplus S$	Partition Number	$T_{cand}(s)$	$T_{quad}(s)$	$T_{total}$	Speedup
$parks \oplus sports$	8192	1.84	0.55	2.39	361.56
$parks \oplus sports$	16384	1.77	0.55	2.32	337.08
$parks \oplus sports$	32768	1.75	0.56	2.31	412.51
$parks \oplus sports$	65536	1.77	0.59	2.36	511.17
$roads \oplus parks$	8192	50.23	10.33	60.56	345.72
$roads \oplus parks$	16384	50.56	10.32	60.88	340.93
$roads \oplus parks$	32768	50.24	12.78	63.02	348.27
$roads \oplus parks$	65536	50.19	15.90	66.09	314.06

**TABLE 3.4** ADP-GPU execution time for different pairs of datasets on an Nvidia Titan-V

### 3.7.4 GPU Speedup for ADP

Table 3.4 shows the time it takes for using CUDA to perform ADP on different pairs of datasets to generate a given number of partitions.

$T_{cand}$  is the time used for finding the candidates, which is a combination of Algorithm 6 and Algorithm 7. For the same pair of datasets,  $T_{alg1}$  doesn't change as the number of candidates doesn't change much when the partition number changes.  $T_{quad}$  is the time for implementing Quadtree partitioning.

When the size of the two datasets increases, the time for finding candidate pairs grows faster than the Quadtree partitioning time. This is shown in Table 3.4. The time needed for Quadtree partitioning also increases with increased in size of data. Compared with sequential ADP in Table 3.3, our GPU implementation on NVidia Titan V provides a stable speedup between 314 to 511 times. The GPU here has equivalent performance with 11 to 18 normal computing nodes (28 cores each node) on *Bridges*.

The drawback of this CUDA implementation is the memory limit of a GPU. Not only the MBRs and weights of two input datasets need memory space, the generated candidates and their weights also need additional GPU memory. For instance, the GPU we used has 12 GB memory, which allows a single run of ADP-GPU to partition *roads* and *parks* at most. For any larger pair, multiple runs of ADP-GPU are required.

### 3.7.5 Weak scaling for ParADP

Here we discuss weak scaling experiments for ParADP. We generate new pairs of datasets by duplicating geometries in *parks*. When the number of compute nodes increases by 16, one duplication of *parks* is added to the workload and the number of cells in the target grid increases by 8192. As shown in Table 3.5, ParADP has good weak scaling.

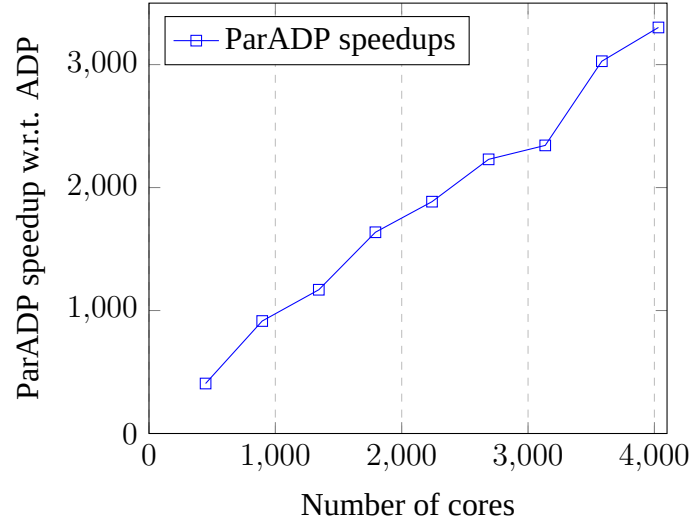
R	S	Candidates	Nodes	Grid cells	$T_{total}(s)$
<i>roads</i>	<i>parks</i>	75 M	16	8192	49.32
<i>roads</i>	2* <i>parks</i>	150 M	32	16384	38.63
<i>roads</i>	3* <i>parks</i>	225 M	48	24576	36.89
<i>roads</i>	4* <i>parks</i>	300 M	64	32768	41.45
<i>roads</i>	5* <i>parks</i>	375 M	80	40960	40.13
<i>roads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

TABLE 3.5 ParADP execution time for weak scaling

### 3.7.6 Strong scaling for ParADP

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

In Table 3.6,  $T_{total}$  stands for the total time for ParADP and  $T_s$  stands for the time for parallel sorting step. In all instances, ParADP has high speedups as shown in Figure 3.8. ParADP has high efficiency which ranges from 0.84 to 1.02, where the highest efficiency of 1.02 is achieved with 32 nodes. The reasons that the efficiency is greater than 1 are that 1) with data decomposition for both layers, the query range for every geometry is sharply reduced; 2) within a certain number of nodes, the parallel sorting time decreases with more nodes as  $R$  doesn't change.



**FIGURE 3.8** Speedups of ParADP w.r.t. ADP for generating a grid with 65536 cells using two datasets - 1) *roads* (72 million polylines) and 2) *parks* (10 million polygons).

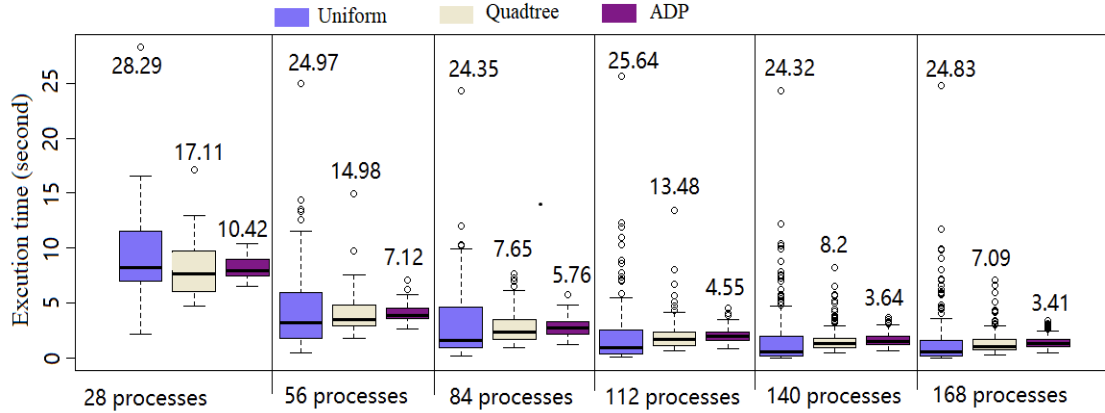
$R$	$S$	Partition Number	Nodes	$T_{total}(s)$	$T_s(s)$
roads	parks	65536	16	55.56	1.82
roads	parks	65536	32	24.69	0.73
roads	parks	65536	48	19.32	0.64
roads	parks	65536	64	13.80	0.38
roads	parks	65536	80	11.98	0.32
roads	parks	65536	96	10.13	0.15
roads	parks	65536	112	9.64	0.15
roads	parks	65536	128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

**TABLE 3.6** ParADP execution time for strong scaling

### 3.7.7 Partition Quality

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

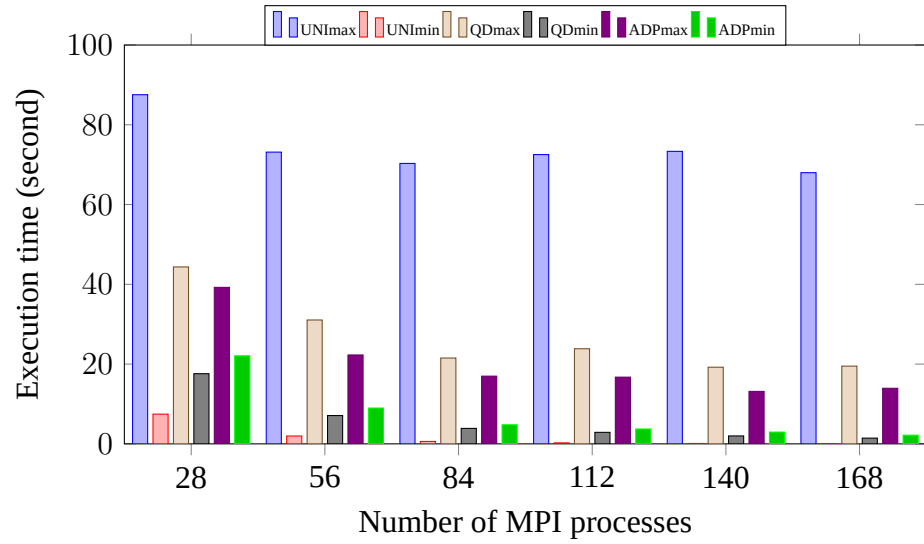
terminates the end-to-end time.



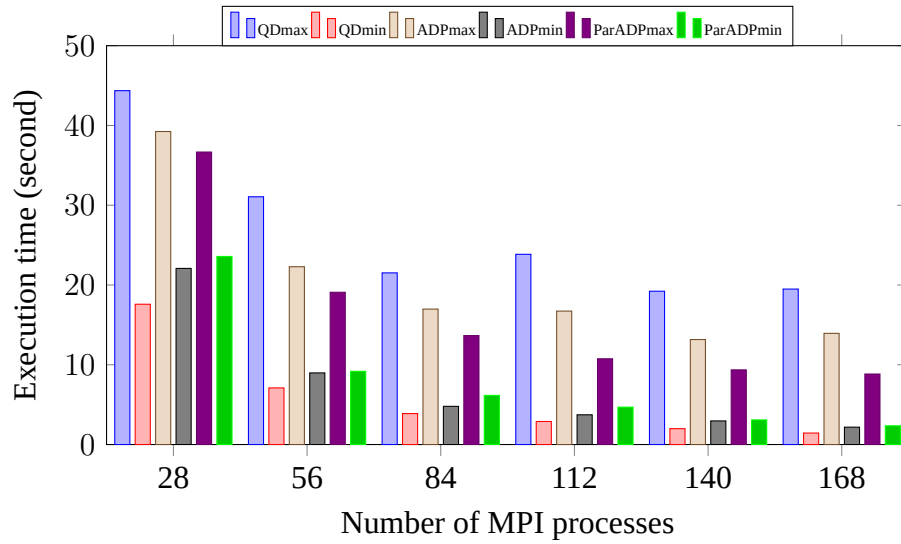
**FIGURE 3.9** Box-plot showing distribution of execution time by different MPI processes running GEOS *Intersects* query using *parks* and the *sports* data. Each time the data sets are partitioned into 8192 parts. Max process execution time along with few outliers are also shown for each partitioning scheme.

Figure 3.9 shows the comparison of execution time for ADP, Quadtree Partitioning, and Uniform Partitioning. As shown in the figure, with more MPI processes involved, the average execution times decrease. However, using Uniform partitioning, the maximum MPI process execution time doesn't change much; using Quadtree partitioning, the overall maximum MPI process execution time changes slightly but in some cases it increases. Since the execution time of a parallel application is decided by the thread taking the longest time (straggler effect), using ADP minimizes the overall execution time.

Figure 3.10 shows the timing for the refinement phase using GEOS *Intersection* method. We compare the partition quality using ADP, Quadtree partitioning, and ParADP. Figure 3.10(a) shows the maximum (max) and minimum (min) MPI process times when an MPI-GIS implementation applied *Intersection* on the partitioned *parks* and *sports*. Both data are partitioned into 8192 parts. As shown in the figure, the max process times for ParADP are much lower than the max process times for Quadtree partitioning. The min process times for ParADP are higher than the min process times for Quadtree partitioning. The MPI process times for ParADP-based partitioned data are in a much narrower range than the MPI process times for Quadtree-based partitioned data. Figure 3.10(b) shows the

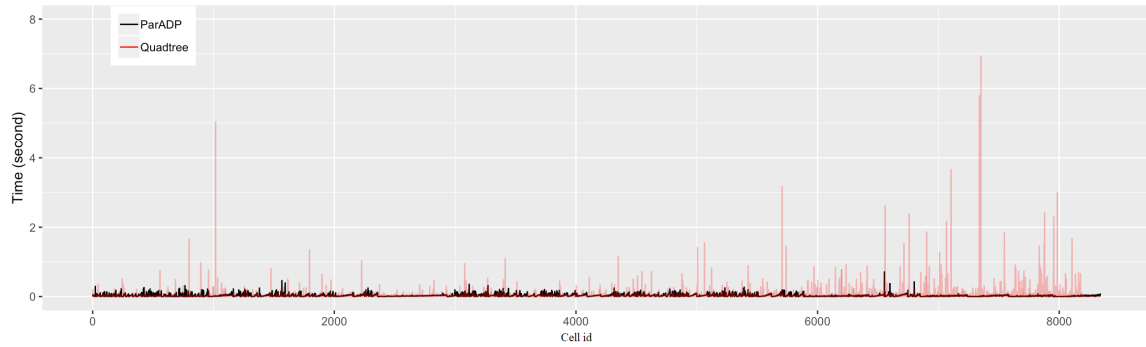


(a) Applying GEOS *Intersection* on two datasets generated by ADP, Quadtree partitioning, and uniform partitioning.

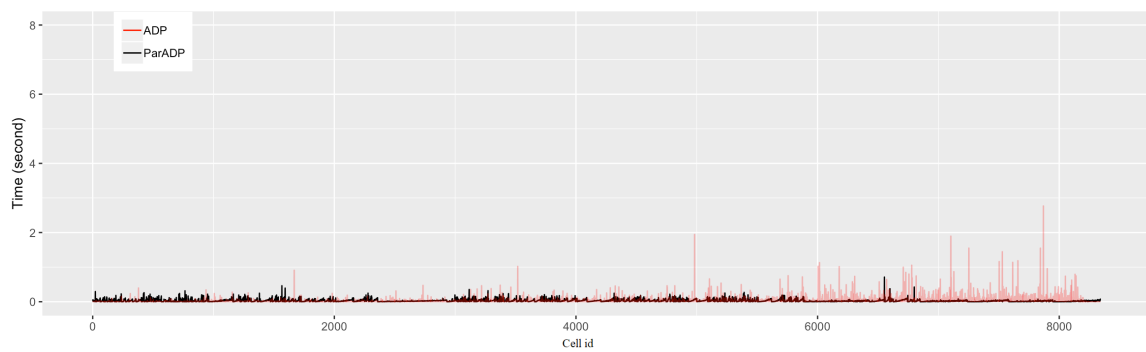


(b) Applying GEOS *Intersects* method on two datasets generated by ADP, ParADP, and Quadtree partitioning.

**FIGURE 3.10** Maximum and minimum process execution times. Datasets *parks* and *sports* were used and partitioned into 8192 parts.



**FIGURE 3.11** Execution time of applying *Intersects* on different cells of the partitioned *parks* and *sports*. The data sets are partitioned into 8192 cells by ParADP and Quadtree partitioning.



**FIGURE 3.12** Execution time of applying *Intersects* on different cells of the partitioned *parks* and *sports* data. The data sets are partitioned into 8192 cells by ParADP and ADP.



maximum (max) and minimum (min) MPI process times when *Intersects* operation is applied on the partitioned *parks* and *sports*. ParADP shows improvement over Quadtree partitioning in both experiments.

The *Intersects* execution times for processing each cell of partitioned *parks* and *sports* are shown in Figure 3.11. The *parks* and *sports* are partitioned to 8192 cells by ParADP and Quadtree partitioning. As we can see, the *Intersects* execution times for processing ParADP-based partitioned cells are within a narrow range and none of them exceed 0.8 second. On the other hand, the *Intersects* execution times for processing Quadtree-based partitioned cells have higher variation and the longest execution time taken is 7 seconds. If we consider a large scale HPC system with as many CPU cores as the number of partitions and each CPU core is assigned data in a pair of cells, spatial join can be done within 0.8 second using ParADP partitioned cells while 7 seconds are needed to process Quadtree-based partitioned cells.

The *Intersects* execution times for processing each cell of partitioned *parks* and *sports* are shown in Figure 3.12. The *parks* and *sports* are partitioned into 8192 cells by ADP and ParADP. Cells with higher cell ID take a longer time in ADP. This is because cells with higher ID have larger weight. Compared to ADP, ParADP shows a narrower process execution time range and a lower value for maximum MPI process execution time.

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

For load balancing spatial computations, an alternative approach is to start with a grid that is based on a single layer (dataset) and dynamically rebalance the workload in cells that have higher workload. In a distributed memory environment, this leads to the movement of complex geometries from an MPI process with higher workload to another MPI process with lower workload. Moreover, there is overhead involved in serializing, deserializing

and parsing the geometries due to communication. This is based on our prior experience of parallelizing spatial join with ADLB library for load balancing. The size of individual geometries varies from few KB to 10 MB. Therefore, the cost of dynamic load balancing while running partition-based spatial join is quite high. Thus, we explored the feasibility of generating a grid with user-specified number of partitions in this paper.

### **3.8 Conclusion**

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

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

### **3.9 Acknowledgment**

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

## CHAPTER 4

### LOAD BALANCING SPATIAL JOIN BY WORK STEALING ON SHARED AND DISTRIBUTED MEMORY

#### 4.1 Introduction

Spatial join is an important operation for analyzing spatial data. Parallelization is essential to accelerate spatial join performance. However, load imbalance due to data skew limits the scalability of parallel spatial join. There are some techniques to address this problem. One of the techniques is to use data and space partitioning to minimize workload differences across threads/processes. However, load imbalance still exists due to differences in join costs of different pairs of input geometries in the partitions. Another technique is to share spatial join tasks among threads using a shared queue.

For the load imbalance problem, we present our parallel spatial join system, WSSJ-DM. WSSJ-DM benefits from balanced partitioning research. Moreover, we experimentally show that our system works well with unbalanced partitioning and spatially un-partitioned datasets, with minor impact on its overall performance. WSSJ-DM uses work stealing technique to share join tasks on shared memory. We study the effect of memory affinity in work stealing operations involved in spatial join in a NUMA-aware system. On distributed memory, non-blocking communications are used to shuffle tasks between busy and idle nodes.

In all the experiments we have done, WSSJ-DM significantly outperformed static and dynamic load balancing methods. WSSJ-DM performed spatial join using *ST\_Intersection* on *Lakes* (8.4M polygons) and *Parks* (10M polygons) in 30 seconds using 35 compute nodes on a cluster (1260 CPU cores). A Master-Worker implementation took 160 seconds in contrast.

This chapter is organized as follows. Section 4.2 introduces background information and Section 4.3 introduces related work. WSSJ is introduced in Section 4.4, and is extended for distributed memory in Section 4.5. Section 4.6 evaluates the performance of WSSJ and WSSJ-DM. Finally, Section 4.7 concludes this work.

## 4.2 Background

In Geographic Information Systems (GIS) and spatial databases, two datasets are combined based on some spatial relationship among geometries in the input datasets. For instance, given two sets of polygons,  $R$  and  $S$ , find all the pairs of overlapping polygons between the two sets, that is, for each polygon  $r$  in dataset  $R$ , find each overlapping polygon  $s$  from dataset  $S$ . This is an example of spatial join [10].

In map overlay, superimposing one dataset containing hurricane swath (polygonal area) and another dataset with county boundaries, is used to determine nearby rescue shelters. Map overlay is the operation where two maps are combined to produce an output map [46]. In spatial analytics, combining two or more datasets gives us insights that are not available in a single dataset. We will focus on spatial join and map overlay problems important in spatial analytics [45].

In a spatial join implementation with a filter-and-refine approach, the computational graph is constructed after loading the geometries from the two input maps and performing spatial indexing and query operations. This graph represents the spatial overlap relationships (many-to-many) from all the geometries of the first map to the geometries of the second map. The resulting relationship graph is a bipartite graph (two disjoint sets for the two input maps) where vertices are geometries and edges represent computational geometry tasks. For real-world maps, the degree of the graph vertices varies widely. This data dependent and irregular variation of degree is one of the factors behind load imbalance. Another factor is variation of the size and density of geometries from one region of the map to another [7, 8].

The overall load imbalance is determined by two factors - 1) size and distribution of geometries in the two input maps that need to be joined together by a process and 2) number of outputs produced per process. The output-sensitive nature makes load balancing difficult because the number of outputs is not known a priori and can not be estimated easily for complex geometries where approximations result in numerous false hits [9, 54]. Therefore, input data and intermediate output data partitioning techniques are used to minimize

variation of load across partitions [10, 28, 29].

An existing approach in partition-based spatial join (PBSM) is to create a certain number of grid cells and assign the cells to processors [14, 20, 29, 46]. Some approaches use static round-robin assignment [20, 29, 46] and others use dynamic load balancing [9, 14]. However, increasing the number of partitions with the number of processors for load balancing leads to increase in problem size due to replication of geometries across partitions [10, 14]. A geometry that overlaps more than one partition has to be replicated. This increases the amount of work (compared to sequential work) as the number of partitions increase [10, 14]. This limits the number of partitions. Our technique for load balancing is more fine-grained because our tasks are at individual geometry level compared to existing approaches that work at grid cell level. Our task construction enables fine-grained load balancing. In our system, the dependency between filter and refine level tasks are handled without exposing the dependencies to the work stealing runtime.

Engineering a work stealing spatial join system on distributed memory is challenging because work stealing requires serialization and communication of complex geometries by a busy sending process, and deserialization (parsing) of geometries at an idle receiving process. This is a significant computation and communication overhead for large geometries. However, this overhead is not present in a shared memory queue based implementation [9]. Another challenge is effective flow control among processes participating in pull-based task sharing in spatial join.

Spatial partitioning for parallel spatial join is challenging with skewed data at scale. Static and dynamic load balancing spatial join systems perform well with balanced partitions. For instance, MapReduce based systems like SpatialHadoop, GeoSpark, and SparkGIS leverage the dynamic load balancing capability of Hadoop and Spark with well-partitioned input data [7, 64, 65]. However, these systems do not support work stealing. This is also the case with MPI-based spatial join systems like MPI-GIS and ParADP [28, 29, 56]. As such, a thread joining partitions with maximum work can become a bottleneck because these systems do not support dynamic task sharing or dynamic re-partitioning. The pro-

posed WSSJ-DM system can handle this scenario at run-time because the idle processes steal work from busy processes in a fine-grained manner.

Our flow control using MPI Remote Memory Access (RMA) guides the granularity and timing of task sharing to keep the idle processes busy and while minimizing the overheads at busy processes. The new design is able to leverage multiple compute nodes efficiently to speedup parallel spatial join, in the presence of serialization and work coordination overheads. From a performance perspective, this is an improvement over shared memory spatial join [9] and distributed memory MPI-based spatial join systems [28, 29, 46].

We present the effect of memory affinity in work stealing operations involved in spatial join on a NUMA-aware system. Our results complement existing line of work on NUMA-aware spatial join [66, 67].

Contributions of the work presented in this chapter are as follows:

- We provide a novel NUMA-aware Work Stealing Spatial Join system (WSSJ) on shared memory. We extended WSSJ to distributed memory (WSSJ-DM).
- We demonstrate effective mitigation of data skew in a fine-grained manner to avoid stragglers (threads taking much longer than others to finish). Both WSSJ and WSSJ-DM are demonstrated to be load balancing and efficient.
- Both WSSJ and WSSJ-DM can perform a variety of spatial relationship joins and spatial overlay joins. Our system can effectively handle data skew in spatially partitioned and un-partitioned datasets.

## 4.3 Related Work

### 4.3.1 Spatial Join

Spatial join [10] involves two spatial datasets  $R$  and  $S$ . There are two types of Spatial join operations: type 1 is to determine spatial relationship<sup>1</sup>, such as *ST\_Within*, *ST\_Intersects*, and other operations; type 2 is to compute overlay area<sup>2</sup>, such as *ST\_Intersection*,

<sup>1</sup>[https://postgis.net/docs/reference.html#Spatial\\_Relationships](https://postgis.net/docs/reference.html#Spatial_Relationships)

<sup>2</sup>[https://postgis.net/docs/reference.html#Overlay\\_Functions](https://postgis.net/docs/reference.html#Overlay_Functions)

*ST\_Union*, and other operations. *ST\_Intersects* is used to answer a query - Is there any overlap between the two geometries? This operation is often faster because the computation can stop as soon as the spatial relationship is confirmed. Join operation of the second type is used in map overlay, and it is more expensive because the entire output geometries have to be computed.

A spatial join on two datasets can be performed in two phases: 1) filtering phase and 2) refinement phase. In the filtering phase, the minimum bounding rectangles (MBR) of geometries are used to produce a collection of candidate pairs, in which MBRs of two geometries from two datasets overlap.

The refinement phase then removes false positives and produces a set of pairs in relationship join or a set of new geometries in overlay join.

#### 4.3.2 Load Balancing in Parallel Spatial Join

Spatial join parallelization has been discussed in [14,15]. Spatial join was implemented on hypercube architecture of the Connection Machine [17] using Census TIGER/Line data. [20] uses partitioning of universe into tiles (grid cells). The tiles are then assigned to processors in a round-robin fashion. Declustering is proposed as a load balancing strategy in [18,19]. [16] uses bitmaps to determine the number of spatial objects to perform dynamic load balancing. SPINOJA [9] uses object decomposition based declustering to mitigate data processing skew on shared memory. MapReduce-based spatial join systems first create data partitions using various partitioning techniques and then use dynamic load balancing supported by MapReduce implementations like Hadoop and Spark to join grid partitions [7,65].

#### 4.3.3 Work Stealing

Work stealing is a dynamic load balancing strategy [22–26]. This strategy has been used in shared memory and distributed memory [23] load balancing solutions.

Chase-Lev's lock-free deque [24] is an important data structure in many shared-memory work stealing designs. The deque uses a *dynamic-cyclic-array*, which allows: 1) the owner to push and pop elements from the top of the deque, 2) others to perform concurrent lock-

free steal from the bottom of the deque. Nhat’s Work Stealing Queue [25] implementation in C++11 is based on Chase-Lev’s lock-free deque and shows a remarkable performance in benchmarks. We use it in our work stealing implementations. For simplicity, we have referred to Work Stealing Queue as queue.

#### 4.3.4 NUMA

In non-uniform memory access, processor cores have access to local memory and remote memory. Remote memory access is costly compared to local access. There has been some earlier work on NUMA-aware algorithms. [67] discusses an experimental study on enabling NUMA-aware main memory spatial join processing. [66] discusses a systematic approach for efficient in-memory query processing on NUMA systems.

Many NUMA policies can be used on current Linux systems. *MPOL\_DEFAULT*, *MPOL\_INTERLEAVE*, *MPOL\_PREFERRED*, and *MPOL\_BIND* are typically available.<sup>3</sup> These policies can be set by calling a system function *set\_mempolicy*. Our findings on NUMA policies are novel.

#### 4.3.5 RMA and MPI Non-blocking Communication

We have used one-sided (put/get) Message Passing Interface (MPI) functions for communicating data among cooperating. One-sided programming model is referred to as Remote Memory Access (RMA) in MPI. It is suitable for expressing irregular communication patterns that arise while coordinating tasks among processes in distributed memory [27]. Non-blocking MPI functions can be leveraged to overlap communication operations with computational steps of spatial join.

---

<sup>3</sup><https://linux.die.net/man/2/mbind>



## 4.4 Implementation of Work Stealing Spatial Join

### 4.4.1 Work Stealing Queue

A simple work stealing system for spatial join on shared-memory consists of the following steps:

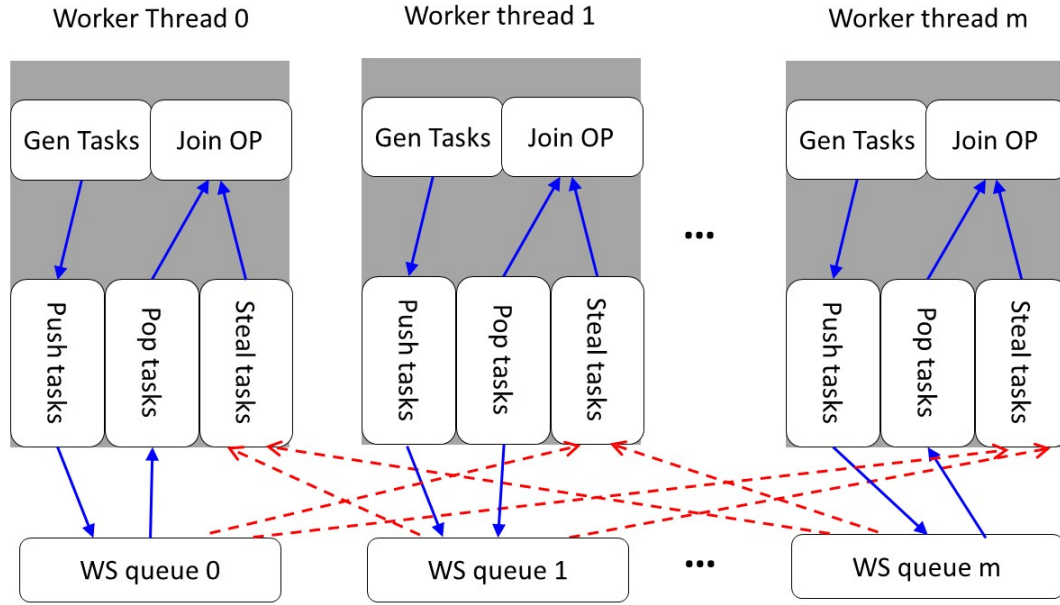
- 1) Create one thread at each processor and each thread uses a queue to hold tasks to be scheduled.
- 2) Each thread pushes its tasks to its own queue from the bottom. And then pops and processes tasks from the queue.
- 3) A thread can steal tasks from the top of other threads' queues after all tasks in its own queue are finished.

Based on these steps, we built a work stealing model for Spatial Join in shared memory (WSSJ) shown as Figure 4.1. In the figure, there are multiple worker threads ( $m + 1$ ) and each worker thread holds its own queue. “Gen Tasks” stands for “Generate spatial join tasks”. “Join OP” refers to Spatial Join operation. A worker thread generates tasks and pushes these tasks into its queue. It can pop tasks from its own queue and steal tasks from other threads' queues.

A worker thread generates tasks and pushes these tasks into its queue. It can pop tasks from its own queue and steal tasks from a victim's queue. The victim can be chosen randomly. In WSSJ, each worker performs the filtering phase and refinement phase independently.

### 4.4.2 NUMA Memory Policies

The execution of spatial join computations are impacted by NUMA memory policies. Spatial join algorithms allocate a temporary buffer to carry out intermediate steps of join algorithm on two geometries. The spatial objects are copied to the temporary buffer to carry out Quadtree partitioning of an individual geometry, to order the coordinates, and to populate the intersection matrix.



**FIGURE 4.1** The Work Stealing Spatial Join model in shared memory (WSSJ). The blue arrows show the direction of flows of Spatial Join tasks within a worker thread. The red arrows show the direction of flows for stolen tasks.

The default NUMA policy on most Linux systems after boot-up is *MPOL\_DEFAULT*, which is “local allocation”. Under this policy, Linux will attempt to satisfy memory requests from the nearest NUMA node of the CPU which submits the memory requests. *MPOL\_DEFAULT* works fine in many scenarios. However, in terms of work stealing, a thread on one NUMA node can steal a task from another NUMA node. A page in memory is accessed by multiple threads. For spatial join, in all experiments we have conducted so far, the tasks on a few worker threads (usually 1 to 4) take much longer to finish than the rest of the threads. When multiple threads allocate and write to temporary buffers for tasks from remote NUMA nodes, there can be memory requests congestion.

*MPOL\_BIND* and *MPOL\_PREFERRED* can mitigate the memory requests congestion issue. Under these two policies, the temporary buffers are on the same NUMA node as the pairs of geometries to be joined. The issue with *MPOL\_BIND* is that it is a strict policy; the OS can only utilize the memory on specified NUMA node(s).

Under *MPOL\_INTERLEAVE* mode, the memory allocations are uniformly distributed among all NUMA nodes. The temporary buffers are allocated in an interleaved manner as

the pairs of geometries are joined.

The NUMA effects discussed here are due to work stealing inherent in parallel spatial join with higher number of threads. We compared the impacts of different memory policies in Section 4.6.1.

#### 4.4.3 Algorithm

Combining the ideas expressed in Section 4.4.1 and 4.4.2, we here present our final NUMA-aware Work Stealing algorithms for spatial join.

Algorithm 9 describes the way that a worker thread generates spatial join tasks and push those tasks into its Work Stealing queue.  $R$  and  $S$  stand for two spatial datasets to be joined. WSSJ uses spatially partitioned datasets. For instance, spatial partitioning of  $R$  and  $S$  into 4 partitions will result in grid cells  $R1$  to  $R4$  and  $S1$  to  $S4$ . This creates 4 join tasks,  $(R1, S1)$ ,  $(R2, S2)$ ,  $(R3, S3)$ ,  $(R4, S4)$ . Each thread is assigned one or more partition(s) as input.  $queues[T]$  are instances of Work Stealing Queue, where  $T$  equals the number of worker threads.

**Task Construction:** A spatial join task consists of a subset of geometries from  $R$  and  $S$  that spatially overlap. We chose one geometry from  $R$  and multiple geometries from  $S$  as a unit task in our system. For overlap detection using minimum bounding rectangle (MBR) approximation of geometries, we use a search tree (index) for MBR query. Assuming, a join on partition pair  $(R1, S1)$ , where  $R1 = \{r_0, r_1, \dots, r_m\}$  and  $S1 = \{s_0, s_1, \dots, s_n\}$ , a unit task is a key-value pair, where key is  $r_i$ ,  $i \in \{0, m\}$  and value is an arbitrary subset from  $S1$ , e.g.,  $\{s_0, s_2, \dots, s_k\}$ , returned by query operation (Line 8).

The *Break\_Down\_Task()* function splits a large task into a set of smaller tasks by breaking down the value part of the task. We set a  $Threshold_{task}$  as the size limit of a task. This step is necessary as a huge geometry  $r$  usually returns a large query result in Line 8 of Algorithm 9, which is one reason of load imbalance.

In WSSJ, each thread occupies one CPU and Algorithm 9 is executed per thread independent of other threads.

---

**Algorithm 9** Algorithm for Pushing Jobs into Queues

---

```

1: Input: Subsets of spatial objects from  $R$  and  $S$ .
2: Output: Queue  $queues[T]$  populated with tasks.
3: Assign NUMA policy.
4: Initialize all the queues in  $queues[T]$ .
5: for Thread  $t_i$  in  $Threads$  do
6:   Build an index  $Index_i$  using MBRs of  $R$ 
7:   for Object  $s_j$  in  $S$  do
8:      $tasks \leftarrow Index_i.query()$ 
9:      $sub\_tasks \leftarrow Break\_Down\_Task(tasks)$ 
10:     $queues[i].push(sub\_tasks)$ 
11:   end for
12: end for
13:

```

---



---

**Algorithm 10** Algorithm for Work Stealing Spatial Join

---

```

1: Input: Queue  $queues[T]$  populated with tasks.
2: Output: Spatial Join  $results$ .
3: for Thread  $t_i$  in  $Threads$  do
4:   while  $queues[i]$  not empty do
5:      $task \leftarrow queues[i].pop()$ 
6:      $results_i \leftarrow Spatial\_Join\_OP(task)$ 
7:   end while
8:   while Not all  $queues$  empty do
9:      $victim \leftarrow Get\_Victim()$ 
10:    while  $queues[victim]$  not empty do
11:       $task \leftarrow queues[victim].steal()$ 
12:       $results_i \leftarrow Spatial\_Join\_OP(task)$ 
13:    end while
14:   end while
15: end for

```

---

Algorithm 10 describes how to choose a victim and steal tasks from the victim's queue. The *Get\_Victim()* function provides the next available victim to be stolen. It follows a cyclic order, beginning with the caller thread's rank+1. This schedule is simple and robust; it can be as efficient as other schedules in WSSJ. Work Stealing Queues store pointers to tasks.

A worker thread  $t_i$  first pops tasks from its own queue  $queues[i]$ , and perform join operations until its queue becomes empty. Then it finds a victim thread. The thief thread will keep stealing and performing join operations until the victim's queue becomes empty. All join results generated by  $t_i$  are pushed into  $results_i$ .

#### 4.4.4 Handling Partitioned and Un-Partitioned Datasets

Generally, parallel spatial join implementations use spatially partitioned datasets. Partitioned datasets are useful to reduce data skew in tasks and make it possible to process datasets larger than available memory. On the other hand, spatial dataset partitioning requires extra time and extra storage space. As WSSJ can share tasks among threads, it is feasible to use spatially un-partitioned datasets (smaller than memory limit) directly.

Let each worker in WSSJ take a part of  $R$  and a part of  $S$  as its input. The subsets of  $R$  and  $S$  can be randomly distributed, as long as the mapping relations of all subsets can be assembled back to the same relations mapping  $R$  to  $S$ , as shown in Formula 4.1. As there is no need to consider the spatial localities of geometries in  $R$ , this step can be done at run-time with no additional cost compared with using partitioned datasets.

In Formula 4.1,  $R$  and  $S$  are randomly distributed into  $n$  and  $m$  parts respectively.  $\oplus$  stands for a spatial join operation. We can get the same join results of  $R$  and  $S$  by performing join operations on all pairs of  $R_i$  and  $S_j$ .

$$\begin{aligned}
 R &= R_0 + R_1 + \dots R_n \\
 S &= S_0 + S_1 + \dots S_m \\
 R \oplus S &= \sum_{i=0}^n \sum_{j=0}^m R_i \oplus S_j
 \end{aligned} \tag{4.1}$$

WSSJ using un-partitioned datasets takes slightly longer to finish when compared to partitioned datasets. The benefit of using un-partitioned data is that no data pre-processing is required, which needs extra computing resources and storage space.

#### 4.5 Framework of Work Stealing Spatial Join on Distributed Memory

We introduced a lock-free deque based work stealing spatial join system (WSSJ) in Section 4.4. A work stealing spatial join system was built on distributed memory architecture with WSSJ working on individual compute nodes. We call it WSSJ-DM in short.

Now we will show how to coordinate data movement due to work stealing across nodes and associated flow control. We are going to describe the overall design of WSSJ-DM

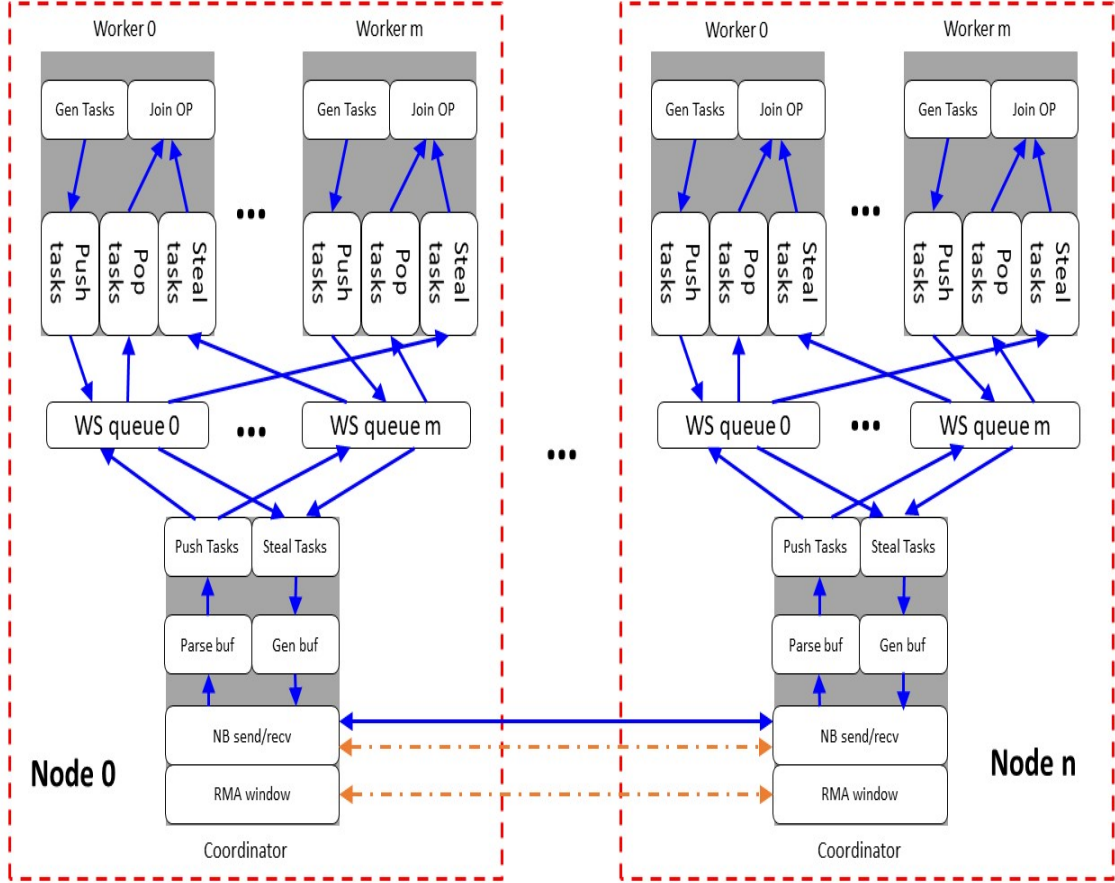
##### 4.5.1 Overall Framework

In our WSSJ-DM design, each node still uses the shared memory work stealing system, plus one coordinator. The coordinators are used to communicate with other nodes and shuffle tasks, as shown in Figure 4.2. A coordinator can spawn multiple threads to speed up the send/rcv procedure.

In Figure 4.2, each dashed red rectangle stands for a WSSJ-DM node. There are multiple nodes. Each node has multiple worker threads, and one coordinator. Each thread has one work stealing deque. The worker threads and dequeues are same as in WSSJ.

In the beginning, each node takes grid cells of  $R$  and  $S$  in a Round Robin manner. A worker thread follows the same flow path as WSSJ: parsing data files, building indices, and pushing tasks into their own work stealing queues. After all the tasks get pushed, the coordinator (*Coord A*) thread begins to monitor its RMA window. If all local queues are empty, *Coord A* begins to seek job from other nodes by writing to the RMA window of other coordinators. Another node (*Coord B*) notices the change in its RMA window. *Coord B* resets its window (to allow new starving coordinator) and begins to steal tasks from local queues and then sends those tasks to *Coord A*.

We discuss the steps in detail in the following subsections.



**FIGURE 4.2** The Work Stealing Spatial Join system on distributed memory (WSSJ-DM). The solid blue arrows show the directions of the flows of spatial join tasks. The dashed orange arrows show the directions of the flows of control messages. “NB send/rcv” stands for “MPI non-blocking send or receive”. “Gen buf” stands for “Generate send buffer” and “Parse received buffer”.

#### 4.5.2 Worker Threads

Same as in WSSJ, the worker threads: 1) build index on their share of whole  $R$  and  $S$  datasets; 2) form spatial join tasks; 3) push tasks into their own deque; 4) pop tasks from their own deque and execute the tasks; 5) steal tasks from other threads in the same node and execute the tasks. Additionally, after all local tasks are finished, the worker threads in WSSJ-DM wait for tasks from their coordinator, which “steal” tasks from other coordinators. The coordinators also inform their worker threads that all tasks across all nodes are done.

The worker threads mainly focus on performing join operations, and behave similarly to worker threads in WSSJ. In the next two subsections, we will discuss the core module of

WSSJ-DM, the coordinators.

#### 4.5.3 Coordinator in Send Status

The coordinators are threads within a WSSJ-DM node meant to facilitate communication with other nodes. A coordinator can be in two status based on the number of all tasks in local work stealing queues: 1) send status and 2) receive status.

A coordinator (*Coord A*) maintains an RMA window, initially as waiting for task requests. It waits until all local spatial join tasks are enqueued. It then steps into the **Send Status**. *Coord A* checks its RMA window periodically. If no change is found, it will update the window with its current remaining tasks and then goes on sleep until next period to save CPU cycles for the worker threads. If there is information that other coordinators are looking for tasks, *Coord A* will mark those coordinators as starving. It then begins to steal tasks from local queues and send those tasks to other coordinators.

The tasks are converted to basic data type arrays to be used by MPI functions. *Coord A* uses MPI non-blocking send function to send those task arrays. It will keep sending tasks to starving coordinators until all local tasks are done or almost done. *Coord A* can fork multiple threads to accelerate the task sending process.

While sending tasks, *Coord A* still checks and updates its RMA window periodically, to be found by other starving processes. *Coord A* also uses MPI non-blocking receive function to gather feedback from other coordinators. If it finds that some coordinators have received too much work to finish in time, it sends a temporary stop sign to those coordinators and stops sending tasks to them. When all local tasks are done, *Coord A* sends a stop sign to all starving coordinators in its record.

#### 4.5.4 Coordinator in Receive Status

After all local tasks are done, *Coord A* enters the **Receive Status**. *Coord A* checks the RMA windows of other coordinators. If a window indicates that all its owner's tasks were finished, *Coord A* records this information and checks the next available RMA window. Among all the other coordinators, it will ask for tasks from the one with the most tasks left



(lets name it *Coord B*). If a window is written by other starving coordinators, *Coord A* will skip this window.

In case when its task request is put on *Coord B*'s window, *Coord A* will use MPI non-blocking receive function to wait for tasks. When the data is received, *Coord A* parses the received data to spatial join tasks and pushes those tasks to an empty queue. This task receiving-parsing-pushing progress can be accelerated by using multiple receive threads. After that, *Coord A* marks the queue to allow workers to steal.

*Coord A* sends the number of its local tasks to *Coord B* after a few invocations of receiving function, also using non-blocking send. *Coord B* uses this number to judge if *Coord A* needs a temporary stop, i.e., *Coord A* has received too many tasks, but its worker threads are processing tasks slowly. If *Coord A* receives a temporary stop, it will be on sleep until most received tasks are done by its worker threads. After waiting, it will again seek another coordinator which still has tasks. If *Coord A* receives a stop sign, it will mark *Coord B* as “all tasks finished” and seek another coordinator for more tasks.

If *Coord A* finds that all other coordinators have no task, it will inform all its worker threads and terminate itself.

#### 4.5.5 Internode Communication

The most important feature of MPI non-blocking send and receive in WSSJ-DM is that it allows overlapping communication and computation.

The coordinators in WSSJ-DM use multiple threads to perform *MPI\_Isend()* to send spatial join tasks and *MPI\_Irecv()* to receive tasks. These *send/receive* threads can perform all send and receive operations concurrently, and then go to sleep. These threads wake up periodically to check if their send and receive operations have finished. Thus, for the most part, send/receive threads are on sleep and yield the CPUs to the worker threads to perform compute-intensive join operations.

Remote Memory Access (RMA) allows access to remote memory. By using the feature, a coordinator in WSSJ-DM nodes can show the node's status in its RMA window. It can tell

others if current node: 1) has spatial join tasks and the number of tasks, or 2) has no tasks, or 3) is hand shaking with another node. A coordinator can also write a request to another coordinator's window based on the information on that window, and wait for instructions for following tasks shuffling.

#### 4.5.6 Theoretical Analysis

We analyze the theoretical performance of WSSJ-DM in this section. The benefit to be gained by WSSJ-DM depends on the computational complexity of spatial join operations because there is a tradeoff between doing work locally vs sending the work to a remote node. For instance, spatial overlay join is more compute-intensive than relationship join. This difference will impact work stealing.

A model is developed here to study the impact of work stealing by remote compute nodes on execution time. Even though multiple processes are active in parallel work-sharing in WSSJ-DM (some in stealing mode and others in victim mode), our model considers one such scenario, to show the scalability bottlenecks because of overheads in work stealing.

Let us assume, among  $n$  nodes, only  $Node_1$  has tasks which require a total of  $V$  computations and the other  $n - 1$  nodes have no tasks. The computing capacity of  $Node_i$  is  $f_i$  which means number of computations done per second.

$f_{i1}$  is the ability of  $Node_i$  to finish tasks that belongs to  $Node_1$ , which is bounded by  $f_i$ , the network, and send/receive buffer generation and parsing speed.

When  $Node_1$  sends tasks to a new node  $Node_i$ , the total computing ability of  $Node_1$  and  $Node_i$  is  $f_1 + f_{i1}$ , minus the cost  $\gamma$  for moving tasks to  $Node_i$ .  $\gamma$  is an average cost which is based on the average size of tasks, buffering of geometries, parsing speed, and the network bandwidth.  $\gamma$  may increase with more idle nodes requesting  $Node_1$  for tasks. We derive Formula 4.2 which models the execution time of WSSJ-DM on tasks consuming  $V$  computing resources before  $Node_1$  reaches its limit of sending tasks.

$$T = \frac{V}{f_1 + (\sum_2^n f_{i1}) - (n - 1) * \gamma} \quad (4.2)$$

We present Formula 4.3 to model the performance of WSSJ-DM in which  $Node_1$  reaches its limit of sending tasks to  $m$  nodes, where  $m$  is fixed and  $n > m$ .

$$T = \frac{V}{f_1 + (\sum_2^m f_{i1}) - (n - 1) * \gamma} \quad (4.3)$$

From Formula 4.2 and 4.3, we can get the following observations about WSSJ-DM:

- 1) WSSJ-DM can always get benefit from additional compute nodes before reaching its bottleneck.
- 2) WSSJ-DM will be slowed down by using more nodes after reaching its bottleneck.
- 3) When  $\gamma$  is high due to a slow or congested network, WSSJ-DM performance will not improve by adding more compute nodes.
- 4) Execution time can be reduced by better compute capacity (higher  $f_{i1}$ ) and faster network (smaller  $\gamma$ ).

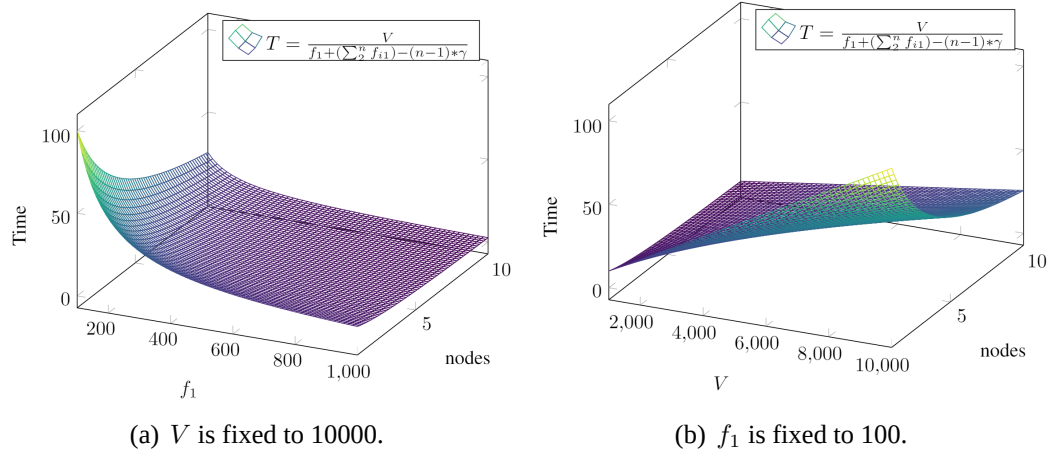
In Figure 4.3(a), we assume:  $V = 10000$ , all  $f_{i1} = f_1/3$ . For convenient, we let  $\gamma = 1$ . The domain of  $f_1$  is  $[100, 1000]$ . The domain of  $n$  is  $[1, 10]$ . We can find that, under the given limit, i.e.  $Node_1$  does not reach its limit of sending tasks, the improvement of more nodes is more noticeable with lower  $f_1$ . More nodes can always bring some improvement.

To explain Formula 4.2, we drew Figure 4.3(b). We assume  $\gamma$  is fixed here. In Figure 4.3, we assume:  $f_1 = 100$ , all  $f_{i1} = f_1/3$ ,  $\gamma = 1$ . The domain of  $V$  is  $[1000, 10000]$ . The domain of  $n$  is  $[1, 10]$ . We can see that, under the given limit, the larger the volume of tasks  $V$  is, the improvement of additional nodes is more significant.

## 4.6 Experimental Results

All of our experiments used five real world datasets: *cemetery*, *sports*, *lakes*, *parks*, and *roads*, which are taken from SpatialHadoop website<sup>4</sup>. The datasets are stored in Well Known Text (WKT) format and the characteristics of these datasets are shown in Table 4.1.

<sup>4</sup><http://spatialhadoop.cs.umn.edu/datasets.html>



**FIGURE 4.3** Theoretical performance modeling of WSSJ-DM before reaching bottleneck.

Name	Type	#Geometries	File size
<i>cemetery</i>	Polygons	193 K	56 MB
<i>sports</i>	Polygons	1.8 M	590 MB
<i>lakes</i>	Polygons	8.4 M	9 GB
<i>parks</i>	Polygons	10 M	9.3 GB
<i>Roads</i>	Polylines	72 M	24 GB

**TABLE 4.1** Attributes of the datasets

All the experiments are done on a cluster named *Bebop*<sup>5</sup> at Argonne National Laboratory. *Bebop* has 664 regular nodes and other nodes. A regular node on *Bebop* has two Intel Xeon E5-2695v4 (36 cores per node), and 128 GB DDR4 memory. We used GCC 8.2.0, C++ 17, Intel MPI 3.1, and GEOS<sup>6</sup> 3.9.1 in all the following experiments.

A Master-Worker and a Round Robin assignment implementations are used for comparison. Master-Worker is a scheduling strategy for dynamic load balancing. A master is in charge of feeding tasks to idle workers. It has been widely used in shared memory solutions, like SPINOJA [9], and distributed memory solutions, like GeoSpark [64], GeoMesa [68]. Round Robin assignment is a widely used technique where each core/node takes parts of partitioned  $R$  and  $S$  in a cyclic manner, and the cores/nodes finish its work independently [20, 29, 46]. Dense areas are distributed among processors due to Round

<sup>5</sup><https://www.lcrc.anl.gov/systems/resources/bebop/>

<sup>6</sup><https://trac.osgeo.org/geos>

Robin assignment.

For pre-processing, three partitioning methods were used: Uniform, Quad-tree, and Adaptive Partitioning (ADP) [28]. [28] shows that for static load balancing,  $ADP > Quad-tree > Uniform$  in general cases.

In following experiments, the value of  $Threshold_{task}$  is set to 20. The number of send/receive thread is set to 5 and the number of tasks per send/receive is set to 100. We also did various experiments on different  $Threshold_{task}$  and different numbers of send/receive thread.

#### 4.6.1 NUMA

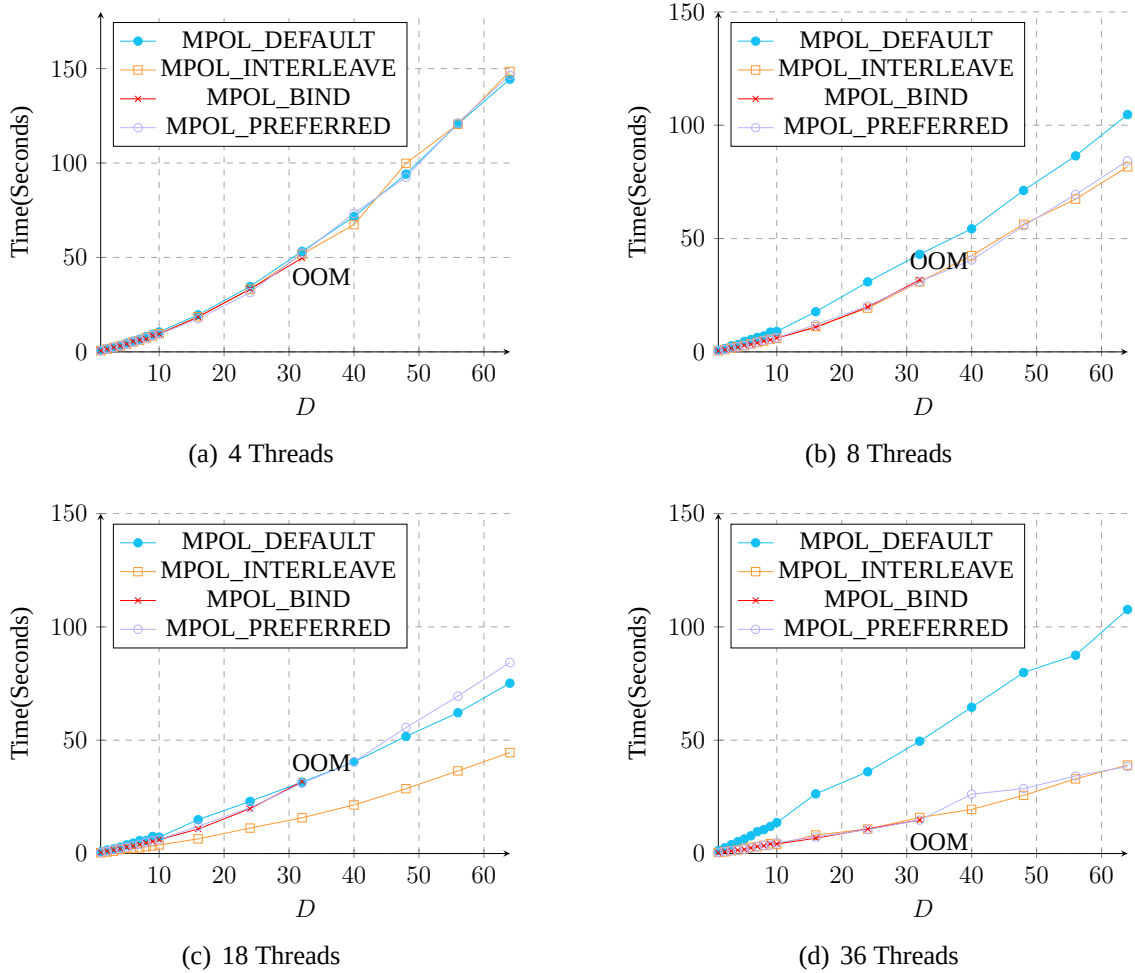
Based on our analysis in Section 4.4.2, we designed comparison experiments among different NUMA policies on WSSJ. The pair of datasets being used is *Sports* and *Cemetery*, which was Quad-tree partitioned into 8192 grid cells. The reason to use *Sports* and *Cemetery* is that both datasets are small and most geometries are small in the datasets. *ST-Intersects* is one of the lightest spatial join operations.

In the experiments, we controlled the sizes of  $R$  and  $S$  by duplicating the original datasets. The duplication coefficient  $D$  means that  $R$  and  $S$  contain  $n$  copies of *Sports* and *Cemetery* respectively.

The regular nodes on *Bebops* only have two NUMA nodes, 0 and 1. The policies settings are: *MPOL\_INTERLEAVE*, node 0 and 1; *MPOL\_BIND*, node 0; *MPOL\_PREFERRED*, node 0; *MPOL\_DEFAULT*. In all tests, threads were evenly distributed on two NUMA nodes.

The results are shown in Figure 4.4. We can see that different policies do not have much difference with 4 threads in Figure 4.4(a). With more threads, in Figure 4.4(d), it takes longer using *MPOL\_DEFAULT* than the other three policies. As we mentioned earlier, more threads may lead to higher memory request congestion between the NUMA domains. The default policy gets negatively impacted by resource contention when compared to other policies.

Because *MPOL\_BIND* only use one NUMA node, it runs out of memory at  $D=40$  while others run out of memory at  $D=80$ .



**FIGURE 4.4** Execution time comparison of different NUMA policies in WSSJ for performing *ST-Intersects* on *Sports* and *Cemetery*.

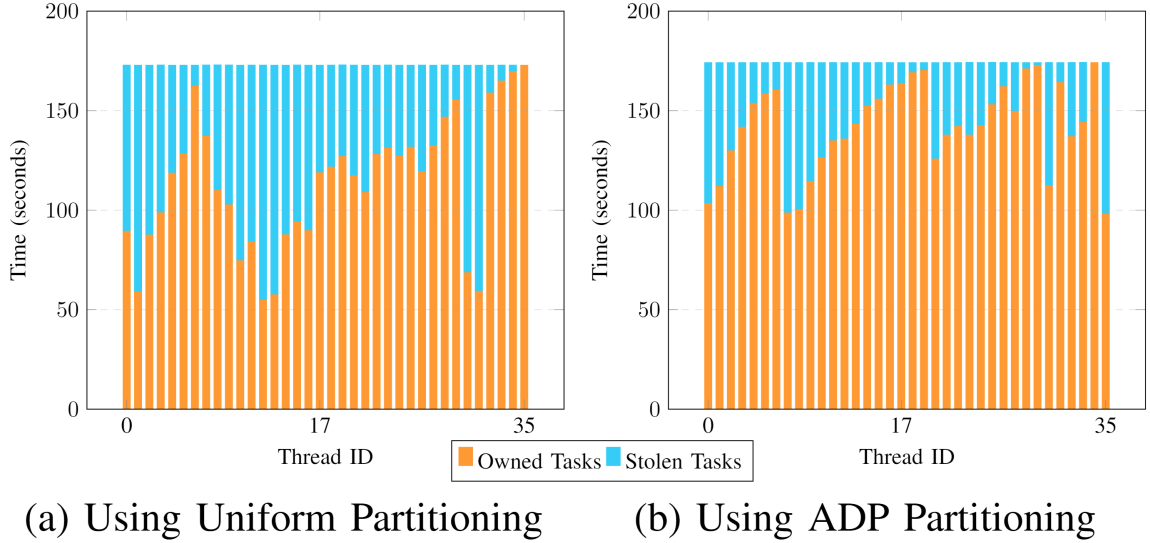
#### 4.6.2 Tasks Composition of WSSJ

There are two types of tasks for a WSSJ worker thread: *owned tasks* and *stolen tasks*. *Owned tasks* are tasks being randomly assigned to a worker in the beginning. *Stolen tasks* are tasks stolen from other workers.

We designed experiments to show the task composition of every WSSJ-DM node in Figure 4.5. We used 36 WSSJ workers (one worker for each core) to perform *ST-Intersection* on *Lakes* and *Parks* which were partitioned into 8192 grid cells using ADP or Uniform

Partitioning.

From Figure 4.5, we can see that the tasks compositions vary in all workers. Every worker was able to finish tasks at approximately the same time. WSSJ is not sensitive to different partitioning approaches. Using Uniform Partitioning is even slightly faster (172s) than using ADP (174s), as it has fewer data duplication (2.38%) than ADP (5.82%).



**FIGURE 4.5** Composition of tasks at different WSSJ workers. Both cases used 36 workers to perform *ST\_Intersection* on *Lakes* and *Parks*.

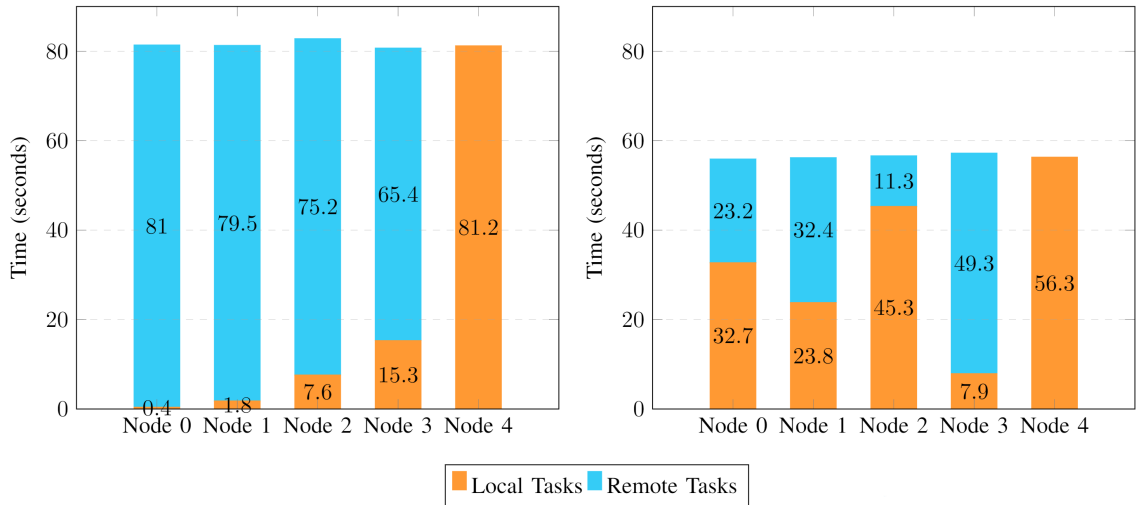
#### 4.6.3 Tasks Composition of WSSJ-DM

A WSSJ-DM node can have two types of tasks: *local tasks* and *remote tasks*. *Local tasks* are tasks being assigned in a Round Robin scheme to each node in the beginning. *Remote tasks* are tasks received from other nodes by its coordinator.

We designed experiments to show the task composition of every WSSJ-DM node in Figure 4.6. We used five WSSJ-DM nodes to perform *ST\_Intersection* on *Lakes* and *Parks* which were partitioned into 8192 grid cells using ADP or Uniform Partitioning.

From Figure 4.6, we can see that the tasks compositions vary in all nodes in both cases. In both cases, there is one node that only works on local tasks. WSSJ-DM is able to re-balance the tasks and manage each node to finish at approximately the same time. We can observe that using a more statically balanced partitioning (ADP) shows a better performance

in WSSJ-DM. This is because a task costs more computing resources being performed remotely than locally. A more balanced initial assignment can reduce the total number of remote tasks.



(a) Using Uniform Partitioning

(b) Using ADP Partitioning

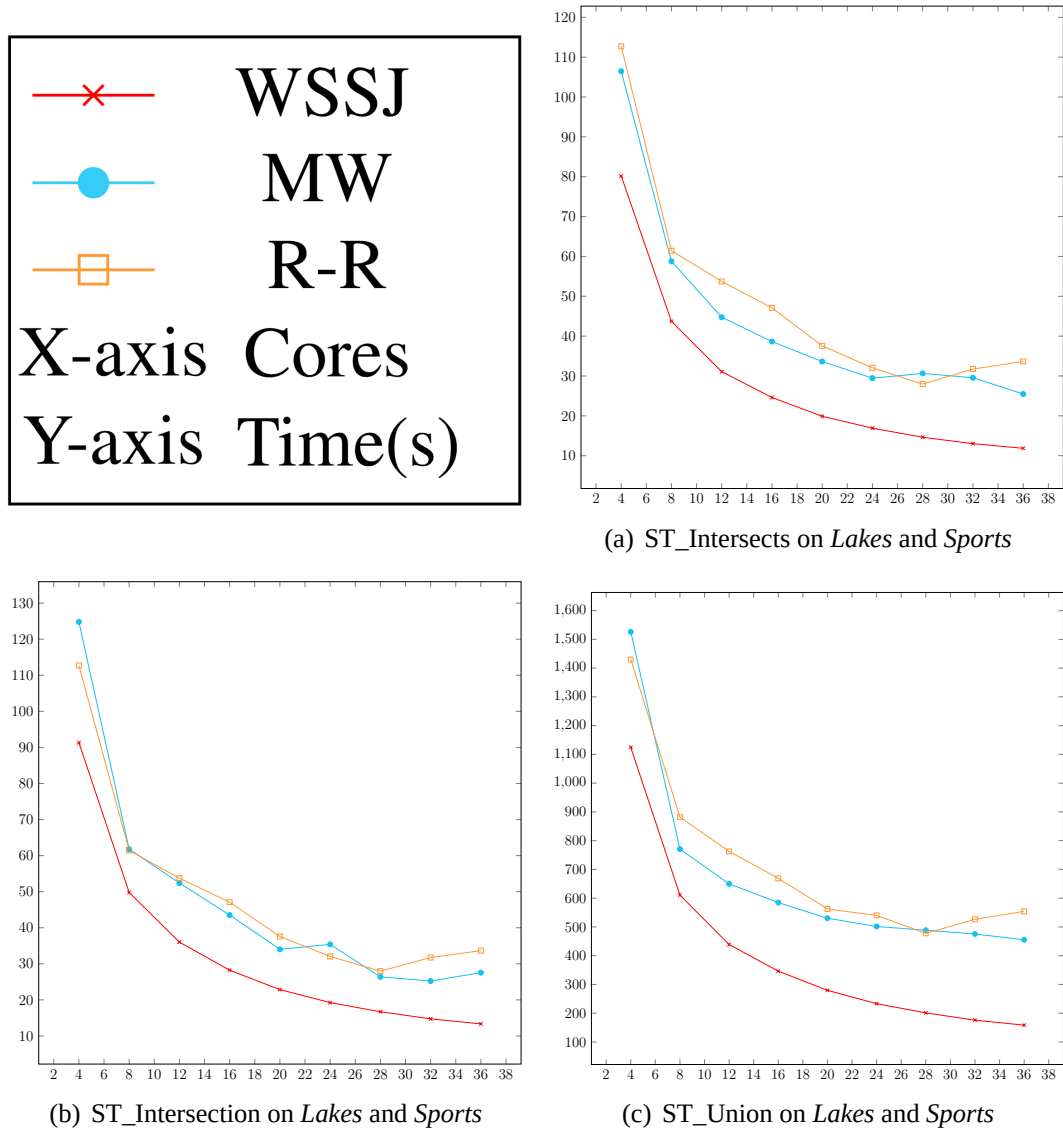
**FIGURE 4.6** Composition of tasks at different WSSJ-DM nodes. Both cases used 5 nodes to perform *ST\_Intersection* on *Lakes* and *Parks*.

#### 4.6.4 Comparison Experiments for WSSJ

We designed experiments to compare the performance of WSSJ, Master-Worker, and Round Robin assignment using different join operations on *Lakes* and *Sports* which were partitioned into 8192 grid cells using ADP partitioning. Round Robin assignment has a better load balancing using ADP partitioning compared with Quad-tree or Uniform partitioning [28].

The results are shown in Figure 4.7. In all cases, a single compute node was used but with different number of cores. WSSJ shows a better performance than *Master-Worker* and *Round Robin assignment* in all cases. In these experiments, WSSJ has a parallel efficiency between 80% (at 36 cores) and 107% (at 4 cores) with respect to sequential spatial join using R-tree index (as shown in Table 4.2).





**FIGURE 4.7** Execution time comparison of using WSSJ, Master-Worker (MW), and Round Robin assignment (R-R) to perform different spatial joins on *Lakes* and *Sports*, which are spatial partitioned into 8192 sub-sets using ADP partitioning.

#### 4.6.5 Comparison Experiments for WSSJ-DM

We compared WSSJ-DM with Master-Worker and Round Robin assignment using different join operations on different pairs of spatial data in Figure 4.8. The experiments were using 1 to 10 nodes (36 to 360 CPU cores).

As shown in Figure 4.8, WSSJ-DM performs better than ADLB, SPINOJA-DM, and Round Robin assignment in most tests. WSSJ-DM performs similar with SPINOJA-DM in the *ST\_Union* test for *Lakes* and *Parks*, in which computing is more densely than *ST\_Inter-*

*section* and *ST\_Intersects*. Execution time of WSSJ-DM and SPINOJA-DM keep decreasing with more CPU cores, while generally WSSJ-DM shows a better performance. The ADLB and Round-Robin implementations reach their bottlenecks quickly because of load imbalance.

WSSJ-DM shows a more significant decrease in time for heavier spatial join operations. In general, *Union* operation is computationally more expensive than *Intersection*. *Intersection* operation is more expensive than *Intersects*. This is reflected in the experimental results and our model also predicted the observed performance difference in Section 4.5.6. Spatial Union operation has the most computational work among the three operations. So, WSSJ-DM has a higher performance for *Union*.

#### 4.6.6 Strong Scaling for WSSJ-DM

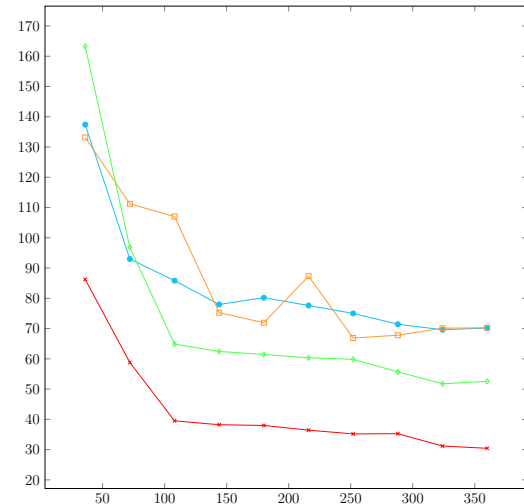
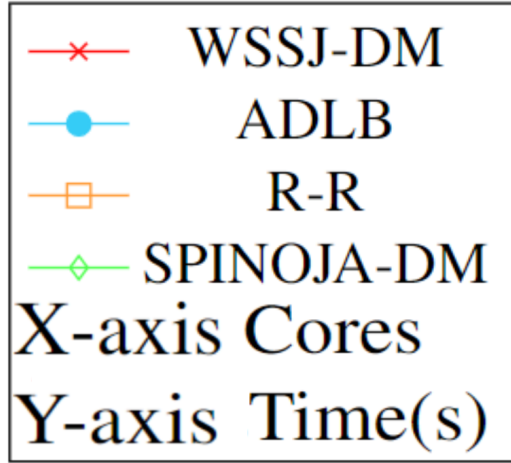
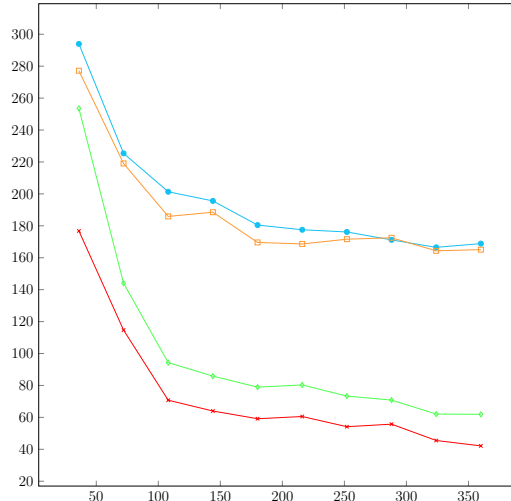
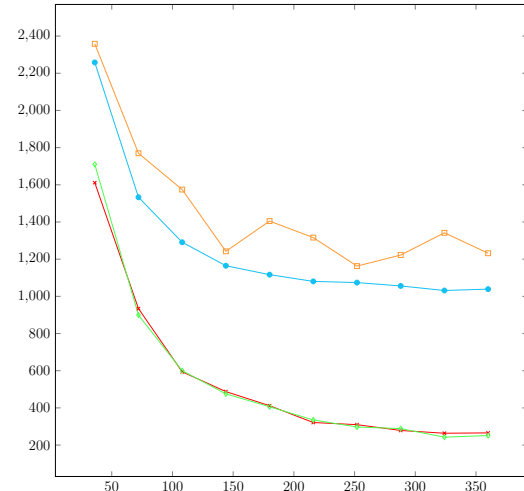
We designed strong scaling experiments for WSSJ-DM. WSSJ-DM was used to perform *ST\_Intersection* on *Lakes* and *Parks* partitioned by different methods. By using different numbers of nodes (36 cores/node), we show the results in Figure 4.9. The corresponding speedups are plotted in Figure 4.9(b).

The results also follow our model that we presented in Section 4.5.6. Due to variation of load across different regions of the input, the performance of WSSJ-DM may fluctuate with different number of nodes. But the general trend is that WSSJ-DM can finish spatial join on *Lakes* and *Parks* faster with more cores before reaching the bottleneck.

WSSJ-DM using ADP partitioning shows the best performance, as ADP is able to provide a better static load balancing than Quadtree or Uniform partitioning [28], which means WSSJ-DM nodes can spend more time on local tasks.

#### 4.6.7 Benchmark

To demonstrate that our system performs well with un-partitioned datasets as well, we used Sequential Spatial Join with Index, WSSJ, and WSSJ-DM to perform *ST\_Intersects*, *ST\_Intersection*, and *ST\_Union* on several pairs of spatially un-partitioned datasets, and the results are shown in Table 4.2. WSSJ was using 1 node (36 cores) and WSSJ-DM was using

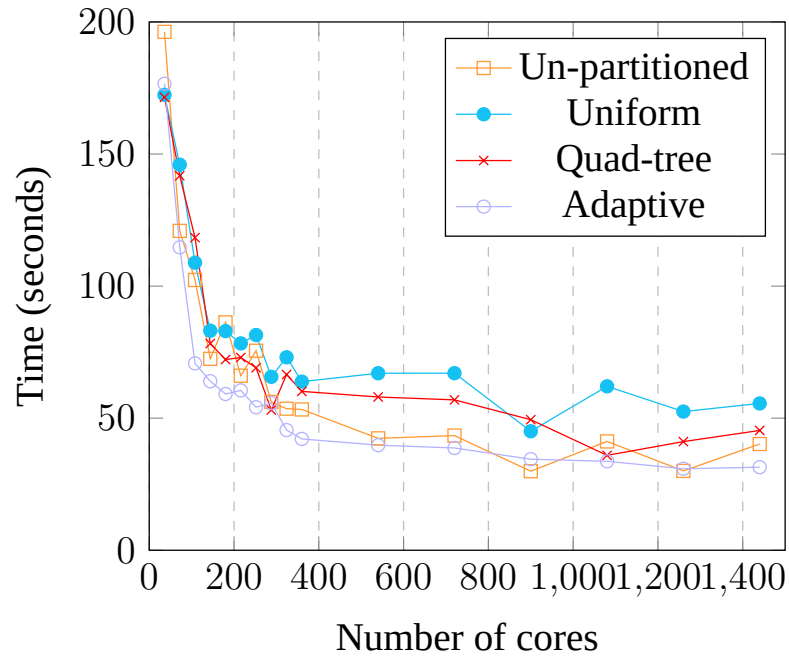
(a) ST\_Intersects on *Lakes* and *Parks*(b) ST\_Intersection on *Lakes* and *Parks*(c) ST\_Union on *Lakes* and *Parks*

**FIGURE 4.8** Execution time comparison among WSSJ-DM, ADLB, SPINOJA-DM, and Round Robin assignment (R-R) performing spatial joins on *Lakes* and *Parks*, which are spatially partitioned into 8192 grid cells using ADP partitioning.

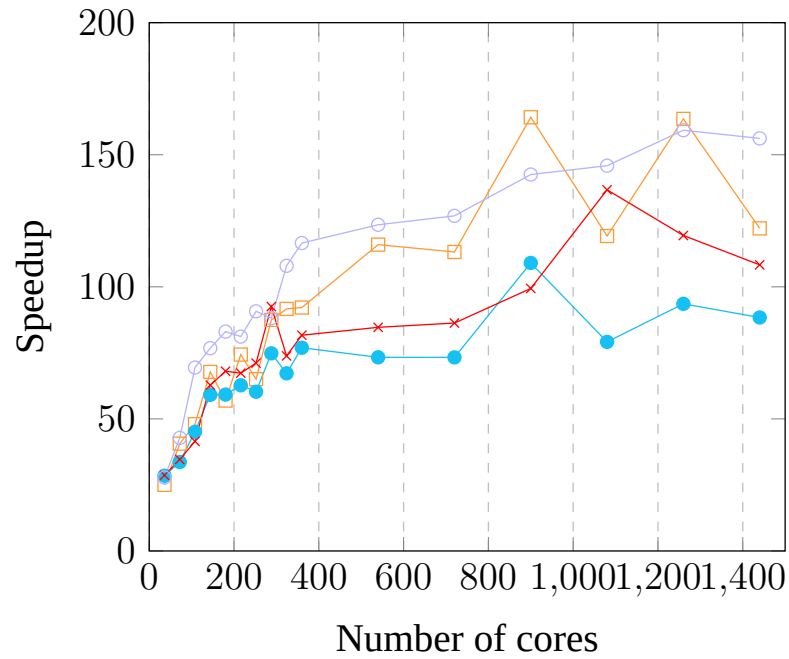
25 nodes (900 cores).

We can see that both WSSJ and WSSJ-DM can be helpful in saving time compared with sequential cases, especially with large datasets. For instance, performing *ST\_Union* on *Roads* and *Lakes* took sequential join 53.45 hours, while WSSJ finished in 1.89 hours and WSSJ-DM finished in 7.26 minutes.

There are two special cases that WSSJ and WSSJ-DM do not show high parallel efficiency, though they are still much faster than the sequential join:



(a) Time (seconds)



(b) Speedup w.r.t sequential spatial join using R-tree index

**FIGURE 4.9** Execution time and speedup plot of WSSJ-DM w.r.t sequential join. For comparison, *ST\_INTERSECTION* was used on *Lakes* and *Parks*. Input data was partitioned into 8192 grid cells using different approaches.

- *ST\_Intersects* on *Roads* and *Lakes*. This is because the memory access becomes bottleneck in this special case. The geometries in *Roads* are usually huge line-strings

while *ST\_Intersects* is a light operation.

- WSSJ-DM on *Sports* and *Cemetery*. These two dataset are small, and the communication (check RMA windows) between different nodes is the major cost in all three cases.

Dataset $R \oplus S$	Join OP	Sequential	WSSJ	WSSJ-DM
<i>Sports</i> $\oplus$ <i>Cemetery</i>	Intersects	3.39	0.59	0.14
<i>Parks</i> $\oplus$ <i>Sports</i>		165.76	10.80	1.78
<i>Lakes</i> $\oplus$ <i>Sports</i>		344.71	16.47	2.90
<i>Lakes</i> $\oplus$ <i>Parks</i>		2,401.74	119.25	20.95
<i>Roads</i> $\oplus$ <i>Lakes</i>		600.60	118.97	20.32
<i>Sports</i> $\oplus$ <i>Cemetery</i>	Intersection	3.92	0.61	0.14
<i>Parks</i> $\oplus$ <i>Sports</i>		339.32	16.14	2.89
<i>Lakes</i> $\oplus$ <i>Sports</i>		389.61	17.546	3.07
<i>Lakes</i> $\oplus$ <i>Parks</i>		4,912.32	196.24	29.92
<i>Roads</i> $\oplus$ <i>Lakes</i>		14,391.57	520.10	35.29
<i>Sports</i> $\oplus$ <i>Cemetery</i>	Union	4.38	0.68	0.13
<i>Parks</i> $\oplus$ <i>Sports</i>		1,908.46	71.82	8.60
<i>Lakes</i> $\oplus$ <i>Sports</i>		4,550.04	179.66	15.49
<i>Lakes</i> $\oplus$ <i>Parks</i>		43,236.40	1,834.39	146.25
<i>Roads</i> $\oplus$ <i>Lakes</i>		192,450.86	6,820.24	435.41

**TABLE 4.2** Execution time (in sec) for Sequential Indexed Spatial Join, WSSJ (36 cores), WSSJ-DM (25 nodes) performing spatial join on different pairs of un-partitioned datasets.

#### 4.7 Conclusion

In this paper, we proposed Work Stealing Spatial Join. WSSJ can perform multiple spatial relationship joins and overlay joins with high parallel efficiency. It can handle skewed data by sharing tasks among workers using work stealing queues. We showed that WSSJ

takes advantage of NUMA policies other than the default policy. We have presented experiments on various real-world datasets and evaluated the performance between WSSJ and two other parallel spatial join methods, Master-Worker and Round Robin assignment. WSSJ has clear advantages on all the tests.

To our knowledge, we introduced the first Work Stealing system for Spatial Join on distributed memory (WSSJ-DM). It uses RMA and MPI Non-blocking communication to shuffle tasks among nodes. Various experiments were conducted on WSSJ-DM. WSSJ-DM shows better performance and scalability than Master-Worker and Round Robin assignment. The results of WSSJ-DM follow the theoretical model we presented.

## CHAPTER 5

### ASYNCHRONOUS DYNAMIC LOAD BALANCING BASED SPATIAL JOIN

We are in the era of Spatial Big Data. Due to the developments of topographic techniques, clear satellite imagery, and various means for collecting information, geospatial datasets are growing in volume, complexity and heterogeneity. For example, OpenStreetMap data for the whole world is about 1 terabyte<sup>1</sup> and NASA world climate datasets are about 17 terabytes<sup>2</sup>. Processing such large data and running spatial analytics require a lot of time. Spatial data volume and variety makes processing and analytics both data-intensive and compute-intensive tasks. In this work, we present spatial data partitioning techniques such as quadtree and uniform grid partitioning based on modeling of spatial join [10] cost. In addition, we present Asynchronous Dynamic Load Balancing (ADLB) [26] based spatial join implementation. The spatial join times modeling experiments expound how geometry collections make load-balancing difficult. We use different spatial data partitioning techniques to find a more balanced method. We evaluate the performance of ADLB-based program by comparing with another MPI-GIS [29] implementation.

#### 5.1 Introduction

With the increasing volume and complexity of spatial data, there is an increasing demand for efficient geospatial techniques for parallelizing spatial computations. This chapter talks about modeling spatial join time, challenges encountered in spatial data partitioning, and bottleneck for handling spatial big data using ADLB to build MPI based GIS. Much of our research on big spatial data has been done on a supercomputer named Bridges at the Pittsburgh Supercomputing Center. Our implementations use Geometry Engine Open-Source (GEOS) library which provides

1) spatial data indices such as Rtree;

<sup>1</sup><https://wiki.openstreetmap.org/wiki/Planet.osm>

<sup>2</sup><https://cds.nccs.nasa.gov/nex>

- 2) geometry-based algorithms;
- 3) and parsing of geometric data.

The datasets used in this chapter are in Well-Known Text (WKT) format, which records geometry objects on a map as a text markup language. For example, a polygon with 3 vertices is represented as `POLYGON((10 20, 30 40, 50 60, 10 20))`. A geometry collection can be represented as `GEOMETRYCOLLECTION(POINT((12 17)), LINESTRING((3 3, -10 10)))`. In section 2, we model polygon intersection costs and reveal one of the causes of load-imbalance in parallel spatial join implementations. In section 3 and 4, we present two contributions of this work which are 1) spatial join cost-based partitioning and 2) asynchronous dynamic load balancing for geospatial computations.

## 5.2 Spatial Data Computing Costs Modeling

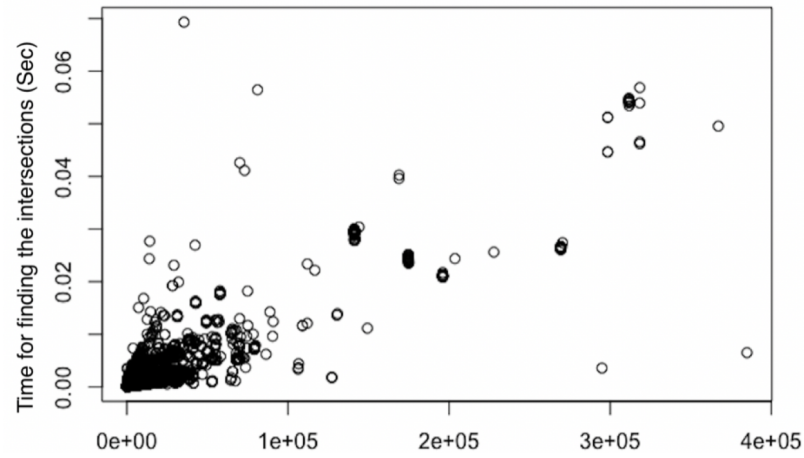
When we try to predict the execution time of spatial join algorithms, there are some special scenarios. The theoretical time complexity for finding whether two geometric shapes with  $n$  and  $m$  vertices intersect is  $O(n*m)$  [69]. However, actual execution time can vary as intersections can be found before going through all the vertices. We perform intersection operation on pairs of geometries which are samples from lakes (8.4 M polygons) and sports (1.8 M polygons).

The execution times for different pairs of geometries are distributed as shown in Figure 5.1(a). After analysis, we figured out that geometry collections cause the distribution to be more scattered. We performed another intersection operation on same pairs of geometries with all geometry collections being split to geometries. The join times for different pairs of geometries are shown in Figure 5.1(b). After geometry collections being divided, the join time shows a better correlation with the theoretical time complexity.

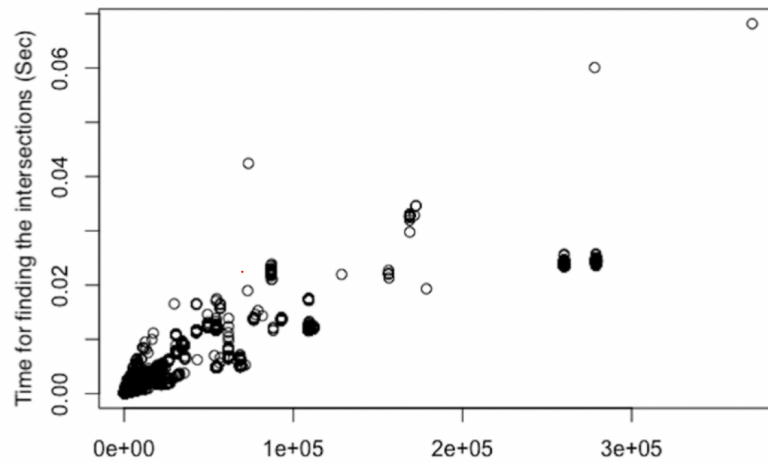
## 5.3 Spatial Data Partitioning

Partitioning spatial data makes spatial computing and dynamic load-balancing much easier. Similar to join operation in databases, we have spatial join operation that is used





(a) Time costs distribution of finding the intersection of two geometries



(b) Time costs distribution of finding the intersection of two geometries, with geometry collection disassembled

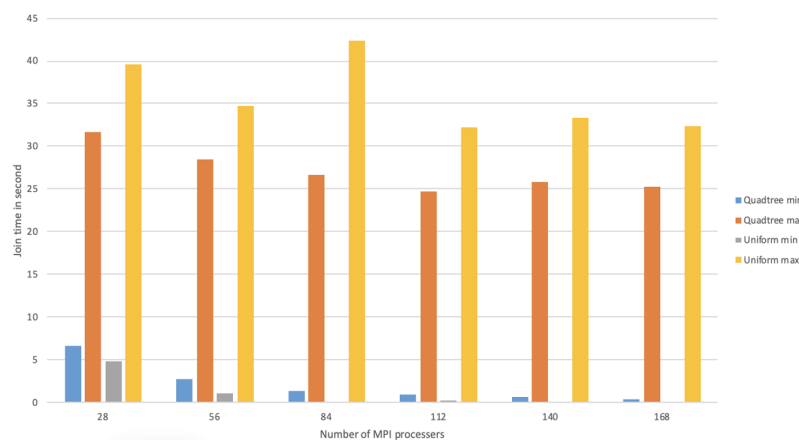
**FIGURE 5.1** The execution time of GEOS to find the intersections between two geometries. In (a) all geometry collections are kept; in (b) all geometry collections are broken down into single geometries in the samples.

in spatial databases and Geographic Information System (GIS). Spatial join finds all-to-all relations between two geometry layers based on whether two shapes overlap or not. With partitioned data, not only the join task is divided into many sub-tasks, but also the spatial query for a single geometry becomes more efficient.

We have embedded our computational cost model inherent in spatial join algorithms to do better partitioning on top of adaptive grid partitioning. This is the novelty in this work. For both quadtree partitioning and uniform partitioning, two spatial datasets, lakes (8.4 M polygons) and sports (1.8 M polygons), are partitioned into 8192 parts. An MPI-GIS implementation performs the join tasks with join tasks being scheduled in round-robin manner to check the quality of different partitioning techniques.

The implementation ran on regular Bridges computing nodes which have two E5-2695 v3 CPUs, i.e. 28 cores each node.

Figure 5.2 shows the performances of two partitioning techniques by comparing the maximum execution times and the minimum execution times of the MPI-GIS program. The maximum execution times for the data partitioned based on quadtree partitioning are 20% to 35% lower than the maximum execution times for the data partitioned based on uniform partitioning. With more processes, the minimum execution time are closer to 0 for both partitioning methods.

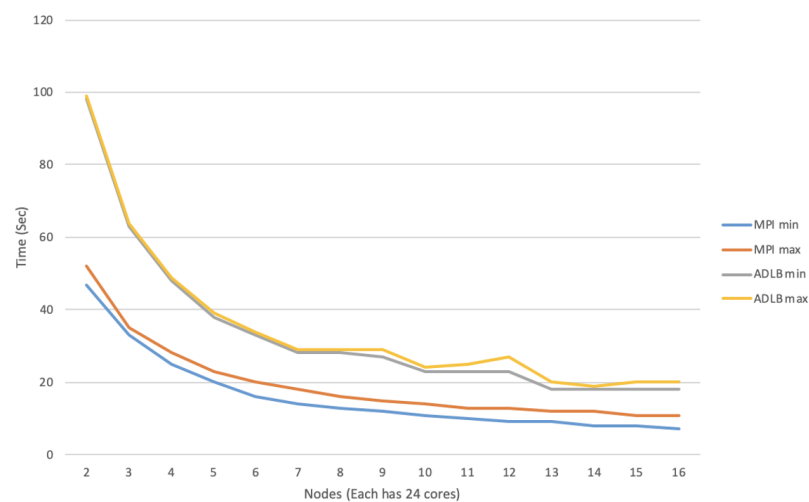


**FIGURE 5.2** Max and min processing times of MPI processes to perform join on two spatial datasets, lakes (8.4 M polygons) and sports (1.8 M polygons).

## 5.4 Dynamic Load-balancing

It's better to use dynamic scheduling with partitioned spatial data. We have built a single-server multi-worker parallel spatial join program for partitioned data. The server keeps the remaining tasks and once a worker becomes idle, it can ask for tasks from the server. The program has been tested using two spatial datasets: Roads (72 million polylines stored) and Sports (1.8 million polygons). The WKT file for Roads is 24 GB and the WKT file for sports is 590 MB. In Figure 5.3, the curve of maximum times (red curve) an MPI process takes is close to the curve of minimum times (blue curve), which indicates the parallel program is load-balanced.

The concept of Asynchronous Dynamic Load Balancing (ADLB) is multiserver multi-worker. We have built an ADLB-based spatial join program. The servers keep join tasks as a group. Once a server runs out of tasks, it asks for tasks from other servers. Once a worker finishes its task, it asks a new task from its server. The same datasets are used for test. In Figure 5.3, the curve of maximum times (yellow curve) an ADLB process takes is closer to the curve of minimum times (grey curve) than the first program. However, the ADLB-based program takes almost double time as the single-server multi-worker program. For ADLB-based spatial join implementation, transferring and parsing data are the overheads.



**FIGURE 5.3** Execution time comparison between two versions: 1) Using ADLB and 2) MPI-GIS dynamic load-balancing for join between Sports (1.8 M polygons) and Roads (72 M polylines).

## CHAPTER 6

### SUMMARY AND FUTURE WORK

#### 6.1 Summary

In Chapter 3, we proposed Adaptive Partitioning techniques. ADP can partition spatial data like polygons and polylines in a load-balanced fashion. We have presented experiments on various real-world data sets and evaluated the partition quality between ADP and two classic partitioning techniques, Quadtree partitioning, and Uniform partitioning. A new duplication avoidance technique is introduced by which unnecessary duplication of geometries spanning multiple grid cells is reduced. OpenMP and GPU versions of ADP was also presented. ADP-OpenMP provides an easy parallization of ADP, and ADP-GPU provides competitive speedup for ADP on a single machine.

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

In Chapter 4, we proposed Work Stealing Spatial Join. WSSJ can perform multiple spatial relationship joins and overlay joins with high parallel efficiency. It can handle skewed data by sharing tasks among workers using work stealing queues. We showed that WSSJ takes advantage of NUMA policies other than the default policy. We have presented experiments on various real-world datasets and evaluated the performance between WSSJ and two other parallel spatial join methods, Master-Worker and Round Robin assignment. WSSJ has clear advantages on all the tests.

To our knowledge, we introduced the first Work Stealing system for Spatial Join on distributed memory (WSSJ-DM). It uses RMA and MPI Non-blocking communication to

shuffle tasks among nodes. Various experiments were conducted on WSSJ-DM. WSSJ-DM shows better performance and scalability than Master-Worker and Round Robin assignment. The results of WSSJ-DM follow the theoretical model we presented.

## 6.2 Future Work

**Scalability of WSSJ-DM:** Chapter 4 introduced a work stealing framework (WSSJ-DM) for spatial join in distributed environment. Though it is faster and efficient than current approaches, it does not scale. Its peak performance reaches at 40 nodes (about 1400 cores).

There is still room to optimize it.

One possible direction is to build a hierarchical data structure for an idle process finding a victim in  $O(\log N)$  time, where  $N$  is the number of processes. From the bottom-up view, let  $n$  computing nodes to form a larger group. And then group  $n$  of such larger groups to be one level higher. A B-tree is formed in this manner. When a starving node tries to seek a busy node to steal from, it first checks its own group, then one level higher. The target seeking process can be done in  $O(\log(p))$  time, where  $p$  is the number of nodes.

One other way is to have each node maintain a  $p$  array, which records all nodes' tasks. For every given unit of time, each node randomly find another node and exchange their task information arrays.

**Utilization of WSSJ-DM for General Purpose:** WSSJ-DM performs on spatial join. It has the potential to be a general load balancer like ADLB. Every component of WSSJ-DM is independent. If we change the spatial join engine to a user define function, or a callback, WSSJ-DM can be used for general purposes.

## Bibliography

- [1] C. Zhang, T. Zhao, L. Anselin, W. Li, and K. Chen, “A map-reduce based parallel approach for improving query performance in a geospatial semantic web for disaster response,” *Earth Science Informatics*, vol. 8, no. 3, pp. 499–509, 2015.
- [2] M.-J. Perles, J. F. Sortino, and M. F. Mérida, “The neighborhood contagion focus as a spatial unit for diagnosis and epidemiological action against covid-19 contagion in urban spaces: A methodological proposal for its detection and delimitation,” *International Journal of Environmental Research and Public Health*, vol. 18, no. 6, p. 3145, 2021.
- [3] H. Samet, Y. Han, J. Kastner, and H. Wei, “Using animation to visualize spatio-temporal varying covid-19 data,” in *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Modeling and Understanding the Spread of COVID-19*, ser. COVID-19. New York, NY, USA: Association for Computing Machinery, 2020, p. 53–62. [Online]. Available: <https://doi.org/10.1145/3423459.3430761>
- [4] W. Qi, R. Procter, J. Zhang, and W. Guo, “Mapping consumer sentiment toward wireless services using geospatial twitter data,” *IEEE Access*, vol. 7, pp. 113 726–113 739, 2019.
- [5] A. Páez and D. M. Scott, “Spatial statistics for urban analysis: a review of techniques with examples,” *GeoJournal*, vol. 61, no. 1, pp. 53–67, 2004.
- [6] R. C. Selley, L. R. M. Cocks, and I. R. Plimer, *Encyclopedia of geology*. Elsevier Academic, 2005.
- [7] A. Eldawy, L. Alarabi, and M. F. Mokbel, “Spatial partitioning techniques in spatial-hadoop,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1602–1605, 2015.

- [8] M. Deveci, S. Rajamanickam, K. D. Devine, and Ü. V. Çatalyürek, “Multi-jagged: A scalable parallel spatial partitioning algorithm,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 803–817, 2015.
- [9] S. Ray, B. Simion, A. D. Brown, and R. Johnson, “Skew-resistant parallel in-memory spatial join,” in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. ACM, 2014, p. 6.
- [10] E. H. Jacox and H. Samet, “Spatial join techniques,” *ACM Trans. Database Syst.*, vol. 32, no. 1, Mar. 2007. [Online]. Available: <http://doi.acm.org/10.1145/1206049.1206056>
- [11] 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.
- [12] J. M. Patel and D. J. DeWitt, “Clone join and shadow join: two parallel spatial join algorithms,” in *Proceedings of the 8th ACM international symposium on Advances in geographic information systems*, 2000, pp. 54–61.
- [13] G. Luo, J. F. Naughton, and C. J. Ellmann, “A non-blocking parallel spatial join algorithm,” in *Proceedings 18th International Conference on Data Engineering*. IEEE, 2002, pp. 697–705.
- [14] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, “Sjmr: Parallelizing spatial join with mapreduce on clusters,” in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–8.
- [15] K. Araki and T. Shimbo, “An mpi-based framework for processing spatial vector data on heterogeneous distributed systems,” in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2016, pp. 554–558.
- [16] S. Shohdy, Y. Su, and G. Agrawal, “Load balancing and accelerating parallel spatial join operations using bitmap indexing,” in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 2015, pp. 396–405.

- [17] E. G. Hoel and H. Samet, "Data-parallel spatial join algorithms," in *1994 International Conference on Parallel Processing Vol. 3*, vol. 3. IEEE, 1994, pp. 227–234.
- [18] K.-L. Tan and X. Y. Jeffrey, "A performance study of declustering strategies for parallel spatial databases," in *International Conference on Database and Expert Systems Applications*. Springer, 1995, pp. 157–166.
- [19] S. Shekhar, S. Ravada, D. Chubb, and G. Turner, "Declustering and load-balancing methods for parallelizing geographic information systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 4, pp. 632–655, 1998.
- [20] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," *ACM Sigmod Record*, vol. 25, no. 2, pp. 259–270, 1996.
- [21] K. Alsabti, S. Ranka, and V. Singh, "An efficient parallel algorithm for high dimensional similarity join," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE, 1998, pp. 556–560.
- [22] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, Sep. 1999. [Online]. Available: <https://doi.org/10.1145/324133.324234>
- [23] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009, pp. 1–11.
- [24] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, 2005, pp. 21–28.
- [25] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli, "Correct and efficient work-stealing for weak memory models," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 69–80, 2013.



- [26] E. L. Lusk, S. C. Pieper, R. M. Butler *et al.*, “More scalability, less pain: A simple programming model and its implementation for extreme computing,” *SciDAC Review*, vol. 17, no. 1, pp. 30–37, 2010.
- [27] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.
- [28] J. Yang and S. Puri, “Efficient parallel and adaptive partitioning for load-balancing in spatial join,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 810–820.
- [29] S. Puri, A. Paudel, and S. K. Prasad, “Mpi-vector-io: Parallel i/o and partitioning for geospatial vector data,” in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–11.
- [30] G. Kedem, “The quad-cif tree: A data structure for hierarchical on-line algorithms,” in *19th Design Automation Conference*, 1982, pp. 352–357.
- [31] H. Samet, “Spatial data structures.” 1995.
- [32] A. Klinger, “Patterns and search statistics,” in *Optimizing methods in statistics*. Elsevier, 1971, pp. 303–337.
- [33] M. J. Berger and S. H. Bokhari, “A partitioning strategy for nonuniform problems on multiprocessors,” *IEEE Transactions on Computers*, vol. 36, no. 05, pp. 570–580, 1987.
- [34] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The r\*-tree: An efficient and robust access method for points and rectangles,” in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, 1990, pp. 322–331.
- [35] B. Seeger, “Performance comparison of segment access methods implemented on top of the buddy-tree,” in *Symposium on Spatial Databases*. Springer, 1991, pp. 277–296.

- [36] J.-P. Dittrich and B. Seeger, “Data redundancy and duplicate detection in spatial join processing,” in *Proceedings of 16th International Conference on Data Engineering* (Cat. No. 00CB37073). IEEE, 2000, pp. 535–546.
- [37] F. P. Preparata and M. I. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.
- [38] E. N. Hanson, “The interval skip list: A data structure for finding all intervals that overlap a point,” in *Workshop on Algorithms and Data Structures*. Springer, 1991, pp. 153–164.
- [39] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, “Efficient processing of spatial joins using r-trees,” *ACM SIGMOD Record*, vol. 22, no. 2, pp. 237–246, 1993.
- [40] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” 2004.
- [41] J. Yu, Z. Zhang, and M. Sarwat, “Spatial data management in apache spark: the geospark perspective and beyond,” *GeoInformatica*, vol. 23, no. 1, pp. 37–78, 2019.
- [42] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, “A case for standard non-blocking collective operations,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer, 2007, pp. 125–134.
- [43] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur, “High performance mpi-2 one-sided communication over infiniband,” in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004*. IEEE, 2004, pp. 531–538.
- [44] W. D. Gropp and R. Thakur, “Revealing the performance of mpi rma implementations,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer, 2007, pp. 272–280.

- [45] 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 *2017 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2017, pp. 232–250.
- [46] S. Puri and S. K. Prasad, “A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system using mpi,” in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 576–585.
- [47] S. Acharya, V. Poosala, and S. Ramaswamy, “Selectivity estimation in spatial databases,” in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, 1999, pp. 13–24.
- [48] 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 9*. Citeseer, 1989.
- [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. Puri and S. K. Prasad, “Efficient parallel and distributed algorithms for gis polygonal overlay processing,” in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*. IEEE, 2013, pp. 2238–2241.
- [51] S. Puri, D. Agarwal, X. He, and S. K. Prasad, “Mapreduce algorithms for gis polygonal overlay processing,” in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*. IEEE, 2013, pp. 1009–1016.
- [52] S. Puri and S. K. Prasad, “Output-sensitive parallel algorithm for polygon clipping,” in *2014 43rd International Conference on Parallel Processing*. IEEE, 2014, pp. 241–250.

- [53] D. Aghajarian, S. Puri, and S. Prasad, “Gcmf: an efficient end-to-end spatial join system over large polygonal datasets on gpgpu platform,” in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016, pp. 1–10.
- [54] Y. Liu, J. Yang, and S. Puri, “Hierarchical filter and refinement system over large polygonal datasets on cpu-gpu,” in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 141–151.
- [55] A. Paudel and S. Puri, “Openacc based gpu parallelization of plane sweep algorithm for geometric intersection,” in *International Workshop on Accelerator Programming Using Directives*. Springer, 2018, pp. 114–135.
- [56] 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 *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pp. 1433–1439.
- [57] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, “Hadoop-gis: A high performance spatial data warehousing system over mapreduce,” in *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 6, no. 11. NIH Public Access, 2013.
- [58] J. Yang, A. Paudel, and S. Puri, “Spatial data decomposition and load balancing on hpc platforms,” in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, 2019, pp. 1–4.
- [59] S. Shekhar, S. Ravada, V. Kumar, D. Chubb, and G. Turner, “Load-balancing in high performance gis: Declustering polygonal maps,” in *International Symposium on Spatial Databases*. Springer, 1995, pp. 196–215.

- [60] M.-L. Lo and C. V. Ravishankar, “Spatial joins using seeded trees,” in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 1994, pp. 209–220.
- [61] J. Zhang, S. You, and L. Gruenwald, “Parallel quadtree coding of large-scale raster geospatial data on gpgpus,” in *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2011, pp. 457–460.
- [62] M. Kelly, A. Breslow, and A. Kelly, “Quad-tree construction on the gpu: A hybrid cpu-gpu approach,” *Retrieved June13*, 2011.
- [63] H. Shi and J. Schaeffer, “Parallel sorting by regular sampling,” *Journal of parallel and distributed computing*, vol. 14, no. 4, pp. 361–372, 1992.
- [64] J. Yu, J. Wu, and M. Sarwat, “Geospark: A cluster computing framework for processing large-scale spatial data,” in *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*, 2015, pp. 1–4.
- [65] 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*, 2017, pp. 1–10.
- [66] P. Memarzia, S. Ray, and V. C. Bhavsar, “The art of efficient in-memory query processing on numa systems: a systematic approach,” in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 781–792.
- [67] S. Ray, C. Higgins, V. Anupindi, and S. Gautam, “Enabling numa-aware main memory spatial join processing: An experimental study,” *ADMS@ VLDB*, 2020.
- [68] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest, “Geomesa: a distributed architecture for spatio-temporal fusion,” in *Geospatial infor-*

*matics, fusion, and motion video analytics V*, vol. 9473. International Society for Optics and Photonics, 2015, p. 94730F.

- [69] L. J. Simonson, “Industrial strength polygon clipping: A novel algorithm with applications in vlsi cad,” *Computer-Aided Design*, vol. 42, no. 12, pp. 1189–1196, 2010.

## Appendices

### A.1 Pseudo code for CUDA quadtree partition

```

1  __global__ void build_quadtree_kernel(Parameters parameters)
2  {
3      // Check the stop criteria.
4      {
5          parameters.weight = 0;
6          for (ulong i = parameters.range_begin + threadIdx.x; i < parameters.range_end; i += blockDim.x)
7              parameters.weight += parameters.in_candidates[i].weight;
8
9          if (parameters.depth >= MAX_DEPTH || parameters.size <= MIN_SIZE || parameters.weight <= MIN_WEIGHT)
10         {
11             final_results->push_back(parameters.MBR);
12             return;
13         }
14     }
15
16     // Count the number of candidates in each child.
17     {
18         // Compute MBRs of the children based on current MBR's center point
19         parameters.node.children.set_bounding_box(parameters.node.center);
20
21         for (ulong i = parameters.range_begin + threadIdx.x; i < parameters.range_end; range_it += blockDim.x)
22         {
23             if (parameters.in_candidates[i].x < parameters.node.center.x
24                 && parameters.in_candidates[i].y < parameters.node.center.y)
25                 parameters.children[0].size++; /* Count on bottom-left child */
26             else if (parameters.in_candidates[i].x < parameters.node.center.x
27                 && parameters.in_candidates[i].y >= parameters.node.center.y)
28                 parameters.children[1].size++; /* Count on top-left child */
29             else if (parameters.in_candidates[i].x >= parameters.node.center.x
30                 && parameters.in_candidates[i].y < parameters.node.center.y)
31                 parameters.children[2].size++; /* Count on bottom-right child */
32             else if (parameters.in_candidates[i].x >= parameters.node.center.x
33                 && parameters.in_candidates[i].y >= parameters.node.center.y)
34                 parameters.children[3].size++; /* Count on top-right child */
35         }
36     }
37 }

```

```

38 // Move candidates
39 {
40     // Compute out_candidates start indexes of each child section
41     ullong start_idx[4];
42     start_idx[0] = parameters.range_begin;
43     start_idx[1] = start_idx[0] + parameters.children[0].size;
44     start_idx[2] = start_idx[1] + parameters.children[1].size;
45     start_idx[3] = start_idx[2] + parameters.children[2].size;
46
47     for (ullong i = parameters.range_begin + threadIdx.x; i < parameters.range_end; range_it += blockDim.x)
48     {
49         if (parameters.in_candidates[i].x < parameters.node.center.x
50             && parameters.in_candidates[i].y < parameters.node.center.y)
51             parameters.out_candidates[start_idx[0]++] = parameters.in_candidates[i]; /* Move for bottom-left child */
52         else if (parameters.in_candidates[i].x < parameters.node.center.x
53             && parameters.in_candidates[i].y >= parameters.node.center.y)
54             parameters.out_candidates[start_idx[1]++] = parameters.in_candidates[i]; /* Move for top-left child */
55         else if (parameters.in_candidates[i].x >= parameters.node.center.x
56             && parameters.in_candidates[i].y < parameters.node.center.y)
57             parameters.out_candidates[start_idx[2]++] = parameters.in_candidates[i]; /* Move for bottom-right child */
58         else if (parameters.in_candidates[i].x >= parameters.node.center.x
59             && parameters.in_candidates[i].y >= parameters.node.center.y)
60             parameters.out_candidates[start_idx[3]++] = parameters.in_candidates[i]; /* Move for top-right child */
61     }
62 }
63
64 // The last thread launches new kernel functions.
65 if (threadIdx.x == QUADTREE_THREADS_PER_BLOCK-1)
66 {
67     // The parameters for 4 children.
68     Parameters c_parameters[4];
69
70     for (int i = 0; i < 4; ++i)
71     {
72         c_parameters.depth = parameters.depth + 1;
73         c_parameters.MBR = parameters.children[i].MBR;
74         c_parameters.size = parameters.children[i].size;
75         c_parameters.range_begin = start_idx[i];
76         c_parameters.range_end = (i == 3) ? start_idx[i+1] : parameters.range_end;
77         c_parameters.in_candidates = c_parameters.out_candidates;
78     }
79
80     // Launch 4 children.
81     build_quadtree_kernel<<<4, QUADTREE_THREADS_PER_BLOCK>>>>(c_parameters);
82 }
83 }
84 }
85

```