8-11-2015

Efficient Parallel and Distributed Algorithms for GIS Polygon Overlay Processing

Satish Puri
Georgia State University

TITLE: EFFICIENT PARALLEL AND DISTRIBUTED ALGORITHMS FOR GIS POLYGON OVERLAY PROCESSING

by

SATISH PURI

Under the Direction of Sushil K. Prasad, PhD

ABSTRACT

Polygon clipping is one of the complex operations in computational geometry. It is used in Geographic Information Systems (GIS), Computer Graphics, and VLSI CAD. For two polygons with n and m vertices, the number of intersections can be $O(nm)$. In this dissertation, we present the first output-sensitive CREW PRAM algorithm, which can perform polygon clipping in $O(\log n)$ time using $O(n + k + k')$ processors, where n is the number of vertices, k is the number of intersections, and k' is the additional temporary vertices

introduced due to the partitioning of polygons. The current best algorithm by Karinthi, Srinivas, and Almasi does not handle self-intersecting polygons, is not output-sensitive and must employ $O(n^2)$ processors to achieve $O(\log n)$ time. The second parallel algorithm is an output-sensitive PRAM algorithm based on Greiner-Hormann algorithm with $O(\log n)$ time complexity using $O(n+k)$ processors. This is cost-optimal when compared to the time complexity of the best-known sequential plane-sweep based algorithm for polygon clipping. For self-intersecting polygons, the time complexity is $O(((n+k)\log n \log\log n)/p)$ using $p \leq (n+k)$ processors.

In addition to these parallel algorithms, the other main contributions in this dissertation are 1) multi-core and many-core implementation for clipping a pair of polygons and 2) MPI-GIS and Hadoop Topology Suite for distributed polygon overlay using a cluster of nodes. Nvidia GPU and CUDA are used for the many-core implementation. The MPI based system achieves 44X speedup while processing about 600K polygons in two real-world GIS shapefiles 1) USA Detailed Water Bodies and 2) USA Block Group Boundaries) within 20 seconds on a 32-node (8 cores each) IBM iDataPlex cluster interconnected by InfiniBand technology.

INDEX WORDS: Polygon Clipping, Polygon Overlay, Parallel Algorithms, MPI, MapReduce, GIS.

TITLE: EFFICIENT PARALLEL AND DISTRIBUTED ALGORITHMS FOR GIS POLYGON OVERLAY PROCESSING

by

SATISH PURI

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

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2015

TITLE: EFFICIENT PARALLEL AND DISTRIBUTED ALGORITHMS FOR GIS POLYGON OVERLAY PROCESSING

by

SATISH PURI

Committee Chair: Sushil K. Prasad

Committee: Ying Zhu

Rafal Angryk

Shamkant Navathe

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

August 2015

DEDICATION

This dissertation is dedicated to my loving and supportive wife, Priti Giri, who has been proud and supportive of my work and who has shared the many uncertainties, challenges, and sacrifices for completing this dissertation. I am truly thankful for having you in my life. This dissertation is also dedicated to my always encouraging parents, Kishor Puri and Piu Kumari Puri. Thank you for all the unconditional love, guidance, and support that you have always given me.

ACKNOWLEDGEMENTS

This thesis is the result of research carried out over a period of many years. During this period, many people supported my work and helped me to bring it to a successful conclusion, and here I would like to express my gratitude. This dissertation work would not have been possible without the support of many people. I want to express my gratitude to my advisor Dr. Sushil K. Prasad for his support and invaluable guidance throughout my study. His knowledge, perceptiveness, and innovative ideas have guided me throughout my graduate study.

I also present my words of gratitude to the other members of my thesis committee, Dr. Ying Zhu, Dr. Rafal Angryk, and Dr. Shamkant Navathe for their advice and their valuable time spent in reviewing the material. I also would like to extend my appreciation to my colleagues in DiMoS research group for all the support and ideas and to everyone who offered me academic advice and moral support throughout my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

- GIS - Geographic Information System
- GPU - Graphics Processing Unit
- CPU - Central Processing Unit
- VLSI - Very Large Scale Integration
- CAD - Computer Aided Design
- PRAM - Parallel Random Access Machine
- MPI - Message Passing Interface
- GH - Greiner Hormann
- MBR - Minimum Bounding Rectangle
- GPC - General Polygon Clipper
- JTS - Java Topology Suite

PART 1

INTRODUCTION

1.1 Polygon Overlay Processing

Scalable vector data computation has been a challenge in Geographic Information System (GIS). One of the complex operations in GIS is polygon overlay, which is the process of superimposing two or more polygon layers to produce a new polygon layer. For instance, one map of United States representing population distribution and another map representing the area affected by hurricane Sandy can be overlaid to answer queries such as "What is the optimal location for a rescue shelter?" In some cases, when a large volume of spatial data is deployed for overlay analysis, it becomes a time consuming task, which sometimes is also time sensitive. GIS scientists use desktop based sequential GIS systems for overlay tasks. The desktop GIS software typically takes hours to perform overlay for large data sets, which makes it useless for real time policy decisions.

Polygon overlay is a geometric operation on two sets of polygons. The polygon overlay combines the input polygons from two different maps into a single new map. The input to binary map overlay are two map layers $L_1 = [p_1, p_2, .. , p_n]$ and $L_2 = [q_1, q_2, .. , q_m]$ where p_i and q_i are polygons represented as x,y co-ordinates of vertices. The output of the overlay operation is a third layer $L_3 = L_1 \times L_2 = [o_1, o_2, ... , o_k]$ represented by k output polygons and this output depends on the overlay operator denoted as $\times$. Overlay operators such as Union, Intersection, etc, determine how map layers are combined. We use the term polygon clipping to refer to geometric set operations on a pair of polygon. Intersection, union, difference, XOR, etc are the geometric set based operators used with a clipping operation. We use filter and refine based approach for polygon overlay. Filtering phase is carried out using uniform-grid and spatial index. Refinement phase is carried out using polygon clipping algorithms. We have also explored parallel algorithms for clipping

a pair of polygons in this proposal. Such algorithms are of interest from a computational complexity standpoint also.

1.2 Motivation

When large volumes of data are deployed for spatial analysis and overlay computation, it is a time consuming task, which in many cases is also time sensitive. For emergency response in the US, for example, disaster-based consequence modeling is predominantly performed using HAZUS-MH, a FEMA-developed application that integrates current scientific and engineering disaster modeling knowledge with inventory data in a GIS framework. Depending on the extent of the hazard coverage, datasets used by HAZUS-MH have the potential to become very large, and often beyond the capacity of standard desktops for comprehensive analysis, and it may take several hours to obtain the analytical results. Although processing speed is not critical in typical non-emergency geospatial analysis, spatial data processing routines are computationally intensive and run for extended periods of time. In addition, the geographic extents and resolution could result in high volumes of input data. For example, Tiger files include 70 Million spatial objects (size of 60GB) of road segments, water features, and other geographic information in USA. OpenStreetMap includes map information from the whole world including road segments, points of interest, and buildings boundaries with a total size of 300GB.

Similar is the case with VLSI CAD data sets where the layout of PCB circuitry is represented as polygons. In VLSI CAD, polygon intersection and union operations are used for design rule checking, layout verification, mask preparation, etc. In an Intel study published in 2010 regarding large scale polygon clipping for microprocessor chip layout, some polygon clippers took several days to run on a one-terabyte memory server [3]. The best polygon clipper at that time took 4.5 hours on a Xeon 3.0 GHz workstation with 32 GB of RAM. As such, the challenges are excessive run-time, excessive memory consumption, and large data size.

We have reviewed the literature extensively on GIS vector overlay computation. What

we have found is that

1. lack of output-sensitive parallel algorithms for polygon clipping,
2. although polygon overlay processing has been regarded as an important class of applications, there is very little reported work showing implementation or performance results,
3. the state-of-the-art is desktop based computing for GIS scientists and those solutions do not employ parallel algorithms.
4. there is no prior work on MPI-based/Hadoop MapReduce based parallel or distributed polygon overlay computation on a cluster.

As multi-core CPUs and many-core GPUs are widely used, the research into parallel techniques and systems on these architectures has become important. However, some of the underlying problems are hard to scale because of non-uniform data distribution and irregular computations and communications in GIS analysis algorithms causing load imbalance. Although there are background literature on parallel/distributed algorithms for polygon overlay computation, there is a lack of an end-to-end comprehensive polygon overlay system on a hybrid cluster with GPUs to the best of our knowledge. The motivation of our system is to provide a general and efficient solution for polygon overlay processing on GPU-based clusters to bring down the running time to within seconds for practical applicability.

1.3 Proposed Research and Development Agenda

We propose to develop distributed system based on MPI and Hadoop running on cluster of computers equipped with Graphics Processing Units (GPUs) as accelerator, which will benefit from multi-core and many-core parallelization of geo-spatial primitives like polygon overlay, buffer computation, etc. We also propose to improve the time complexity using PRAM model for polygon overlay operation by developing output-sensitive algorithms and

implementing them on shared memory machines. We summarize our goals, system design, implementation details, and improved time complexity results.

1.3.1 Goals

1. Output-Sensitive PRAM algorithms for clipping a pair of polygon
   (a) Parallelization of plane-sweep based algorithm.
   (b) Parallelization of graph-traversal based algorithm.
   (c) Complexity analysis of parallel algorithms using PRAM model.
2. Parallel polygon overlay using shared memory
   (a) Pthread based implementation for overlaying two layers of polygons.
   (b) CUDA (GPU) based design and implementation for polygon clipping.
   (c) Java threads based parallelization of Vatti's algorithm and Greiner-Hormann's algorithm.
3. Distributed Polygon overlay design and implementation
   (a) Develop techniques for load balancing, spatial partitioning and spatial indexing using master-slave and peer-to-peer model.
   (b) Message Passing Interface (MPI) based design and implementation.
   (c) Hadoop MapReduce based design and implementation. Develop different versions using single Map phase, Map and Reduce phases and chain of Map and Reduce phases.

1.3.2 Overview of Main Results

With respect to the above-mentioned goals, we have developed *MPI-GIS* and *Hadoop Topology Suite* which are based on MPI and Hadoop respectively. Now, we briefly describe these systems with main results.

1. **Parallel Output-Sensitive Polygon Clipping Algorithms**: Polygon clipping is one of the complex operations in computational geometry. The polygonal data has high degree of irregularity. Some of the U.S. State boundaries consist of about 100,000 polygonal vertices. For two polygons with n and m vertices, the number of intersections can be $O(nm)$. Plane-sweep based polygon clipping algorithm has time complexity of $O((n+k)\log n)$ where k is the number of intersections. Sequential algorithms for this problem are in abundance in literature but there are very few parallel algorithms solving it in its most general form. Using multi-way divide and conquer technique, we have developed the first output-sensitive CREW PRAM algorithm, which can perform polygon clipping in $O(\log n)$ time using $O(n+k+k')$ processors by parallelizing Vatti's algorithm, where n is the number of vertices and k' is the additional temporary vertices introduced due to the partitioning of polygons. The current best algorithm by Karinthi, Srinivas, and Almasi does not handle self-intersecting polygons, is not output-sensitive and must employ $O(n^2)$ processors to achieve $O(\log n)$ time. We have also developed an output-sensitive PRAM algorithm based on Greiner-Hormann algorithm for geometric *union, intersection* and *difference* operations on a pair of simple polygons in $O(\log n)$ time using $O(n+k)$ processors. For self-intersecting polygons, the time complexity is $O(((n+k)\log n \log\log n)/p)$ using $p \leq (n+k)$ processors. This is cost-optimal when compared to the time complexity of the best-known sequential plane-sweep based algorithm for polygon clipping.

   We also presented a new technique based on the fact that if the edges span a bounded region (scanbeam), number of edge intersections can be found out within the region simply by knowing the order in which the edges intersect the boundary of the region. This insight is used to parallelize plane-sweep based on counting the pairs of inversions at each event first and reporting them later for output sensitive processor allocation. Our technique can be easily implemented by parallel sorting as opposed to some of the existing techniques that use complex data structures such as array-of-trees or plane-sweep trees. For deriving logarithmic time complexity, we used parallel segment tree

for efficient partitioning of polygons into scanbeams. Our algorithm handles arbitrary polygons. These insights and algorithmic tools may also be useful for the parallelization of other computational geometry algorithms.

2. **MPI-GIS** We have engineered MPI based distributed system over a cluster of nodes for traditional polygon overlay analysis. We believe MPI-GIS to be the first MPI-based system for end-to-end spatial overlay processing on polygonal data. Earlier version of MPI-GIS was based on master/slave design pattern and employed different architectures; based on static and dynamic load-balancing among worker nodes. For dynamic load balancing, the master process acts as the owner of a pool of overlay tasks from which it assigns tasks to slave processes. The slave processes continuously checks with the master process for new tasks once they finish their individual tasks. The master-slave communication is message-based, handled by MPI send/receive primitives. We enhanced this system by using uniform grid based spatial partitioning and file partitioning for further scalability. *MPI-GIS* achieves 44X speedup while processing about 600K polygons in two real-world GIS shapefiles 1)USA Detailed Water Bodies and 2)USA Block Group Boundaries within 20 seconds on a 32-node (8 cores each) IBM iDataPlex cluster interconnected by InfiniBand technology. It is based on filter and refine strategy. It is able to leverage many-core GPUs for filtering overlapping polygons.

3. **Hadoop Topology Suite** MapReduce paradigm is now standard in industry and academia for processing large-scale data. We have implemented overlay algorithms in Hadoop MapReduce platform in three different forms namely i) with a single map and reduce phase, ii) with chaining of MapReduce jobs, and iii) with a single map phase using *DistributedCache*. We used uniform-grid for spatial partitioning and R-tree for spatial indexing. Based on our experimental results, the first version performed well in case where both data sets are more or less equal in size. However, in case where one of the data sets is smaller in size, then *DistributedCache* based version with a single map phase without shuffle-exchange operation performs the best due to reduced

communication overhead.

PART 2

A SURVEY ON PARALLEL SPATIAL OVERLAY AND JOIN OVER CLOUD AND GPU-ACCELERATED HYBRID PLATFORMS

2.1 Definitions

Polygon Overlay: Polygon overlay is a geometric operation on two sets of polygons. The overlay operation combines the input polygons from two different maps into a single new map. The input to binary map overlay are two map layers $L_1 = [p_1, p_2, .. , p_n]$ and $L_2 = [q_1, q_2, .. , q_m]$ where p_i and q_i are polygons represented as x,y co-ordinates of vertices. The output of the overlay operation is a third layer $L_3 = L_1 \times L_2 = [o_1, o_2, ... , o_k]$ represented by k output polygons and this output depends on the overlay operator denoted as $\times$. Overlay operators such as Union, Intersection, etc., determine how map layers are combined. In case of raster data format, the operation is called grid overlay.

Spatial Join: A type of table join operation in which fields from one layer's attribute table are appended to another layer's attribute table based on the relative locations of the features in the two layers. In contrast to the overlay operation, spatial join takes two sets of spatial records R and S and a spatial join predicate θ (e.g. overlaps) as input, and returns the set of all pairs (r, s) where $r \in R, s \in S$, and θ is true for (r, s), but not the output polygons themselves [4,5]. A typical example of a spatial join is "Find all pair of rivers and cities that intersect." In spatial join, only intersection testing is needed, whereas in polygonal overlay with intersection operator, the new polygon's contour needs to be produced, in addition to the common features.

2.2 Historical Background

Scalable polygonal computation has been a long-standing challenge in the Geographic Information Systems (GIS) community. One of the complex operations in GIS is polygon

overlay, which is the process of superimposing two or more polygonal layers to produce a new layer. For instance, a map of the United States representing population distribution and another map representing the area affected by hurricane Sandy can be overlaid to answer queries such as "What is the optimal location for a rescue shelter?" Many situations of concern involve temporal data including physical phenomenon evolving over time such as solar flares [6], vegetation changes over Sahel region of Africa [7], and weather events, and moving objects such as planets and comets, planes, vehicles, people, troops, etc.

Geo-spatial datasets are large, for example, *Tiger* files of size 60GB include 70M Million spatial objects of road segments, water features, and other geographic information in USA. *OpenStreetMap* includes the map information from the whole world including road segments, points of interest, and building boundaries with a total size of 300GB. Real world polygons can be large in size. For example, one of the polygons representing ocean in shapefile *Ocean* has more than 100K vertices. Geo-spatial computations and analytics are computationally intensive. For example, polygonal overlay using two real-world GIS shapefiles *USA Detailed Water Bodies* and *USA Block Group Boundaries* containing about 600K polygons on ArcGiS 10.1 running on Intel Core i5 processor takes about 13 minutes [8]. The spatial join of a polyline table with 73M records representing the contiguous USA with itself takes roughly 20 hours to complete on an Amazon EC2 instance [9]. Therefore, effectively harnessing parallel processing capabilities of modern hardware platforms is essential.

Similar is the case with VLSI CAD data sets where detailed circuit layout is represented as polygons. In VLSI CAD, polygon intersection and union operations are used for design rule checking, layout verification, mask preparation, etc. In a 2010 Intel study on large scale polygon clipping for microprocessor chip layout, some polygon clippers took several days to run on a server with 1 TB memory [3]. Thus, the challenges are excessive run-time, excessive memory consumption, and large data size.

Efficient parallel and distributed processing of spatial data has been a long-standing research question in GIS community. The irregular and data intensive nature of the underlying computation has impeded the exploratory research in this space. Cloud platform is

well suited for GIS scientists due to web-based accessibility and on-demand scalability. Due to its compute-intensive nature, spatial overlay and join computation requires a high performance approach that can leverage parallelism at multiple levels. In this chapter, we will survey recent advances in parallel and distributed implementations of these computations using high performance and cloud computing technologies.

Previous work related to parallel polygon overlay is surveyed in [10]. Previous work related to parallel spatial join is covered in [5, 11, 12].

2.3 Scientific Fundamentals

In this section, we will describe the significance of cloud computing for GIS applications focussing on Windows Azure and Hadoop MapReduce platform.

2.3.1 Cloud computing

The GIS data is continuously being acquired through a vast majority of sources and hence there is a continuously increasing, overwhelming collection of data repositories (e.g. LIDAR [13]). It has been observed that there is a wide range of large scale distributed computing applications from Geosciences that have significantly varying demand of resources during execution. Allocating a dedicated set of resources, such as a compute-cluster or a Grid, will be underutilized most of the times. On the other hand, the data sets required to be processed during a time sensitive event could demand more resources than those dedicated. Therefore, it is essential for a resource provider to be able to allocate and deallocate resources on demand. For such scenarios Cloud computing proves to be the only viable solution. The challenges behind cloud based solution are i) lacks traditional support for parallel processing and (ii) the tedious exploration of design space for right techniques for parallelizing various workflow components including file I/O, partitioning, task creation, and load balancing.

Since cloud computing offers researchers to think of research questions without worrying about the scale of resources required, it has drawn wide interest from researchers, especially those working with data and compute-intensive scientific applications. Conceptually, the key

idea here is to abstract the provisioning mechanism at a level where users can avail these resources as and when needed without worrying about the scale, availability, or maintenance. Users are billed for utilization, largely based on the time a resource was reserved by a user.

2.3.2 Windows Azure Platform

Windows Azure platform is a cloud computing platform hosted in Microsoft data centers. For computation, there are two types of processes called web role and worker role. The web role acts as a user interface (web application). Windows Azure provides three types of storage mechanisms: Queues, Blobs, and Tables. Storage queues in Windows Azure platform are similar to traditional queue data structure but the first in first out (FIFO) functionality may not be guaranteed. Azure promotes task based parallelism where Queue storage is used as a shared pool to host task IDs for both web role and worker role instances. A reference to a task is usually put as a message on the queue and a set of worker role instances are deployed to process it. The blob storage is a persistent storage service like a traditional file.

2.3.3 Hadoop MapReduce

The cloud computing framework based on Hadoop MapReduce provides a promising solution to solve the problems involving large-scale data sets. The input data is viewed as a stream of records comprising of key-value pairs. As the name suggests, in MapReduce there are two phases namely map phase and reduce phase. A map phase consists of map tasks where each map task can run in parallel and works on a portion of the input. A map task produces intermediate key-value pairs which are processed in parallel by reduce tasks in reduce phase. Under the MapReduce programming model, a developer needs to provide implementations of the mapper and reducer. In Hadoop, both the input and the output of the job are stored in a file-system known as Hadoop Distributed File System (HDFS).

2.4 Key Applications

In this section, we will describe parallelization of spatial overlay and join computation using Windows Azure and Hadoop MapReduce platform. We will also describe the role of high performance computing to accelerate spatial overlay and join.

2.4.1 Polygon Overlay Processing using Windows Azure

The basic idea is to divide the overlay computation tasks among web role and worker roles using Azure queues and Blobs. An Azure Cloud based parallel system for polygon overlay known as Crayons is described in [14, 15].

1. The web role presents the interface with a list of Geographic Markup Language (GML) files available to be processed along with the supported operations (currently union, intersection, xor, and difference). The user selects the GML files to be processed along with the spatial operation to be performed on these files. The web role puts this information as a message on the input queue.

2. Worker roles continuously check the input queue for new tasks. If there is a task (message) in the queue, the worker roles read the message, download the input files, parse them, and create the intersection graph to find the independent tasks. To achieve this, Crayons finds each overlay polygon that can potentially intersect with the given base polygon and only performs spatial operation on these tasks. This is achieved using the coordinates of bounding boxes generated during parsing of input files. Then each worker role shares the tasks it creates among all the worker roles by storing the task IDs in a common task pool.

3. After the workers finish task creation, they fetch work from the task pool and process them by invoking polygon clipping library. After each task is processed, the corresponding worker role permanently deletes the message related to this task from the task pool queue. Additionally, each worker role puts a message on the termination indicator queue to indicate successful processing of the task.

4. The web role keeps checking the number of messages in the termination indicator queue to update the user interface with the current progress of the operation. On completion, the web role commits the resultant blob and flushes it as a persistent blob in the Blob storage. The output blob's uniform resource identifier (URI) is presented to the user for downloading or further processing.

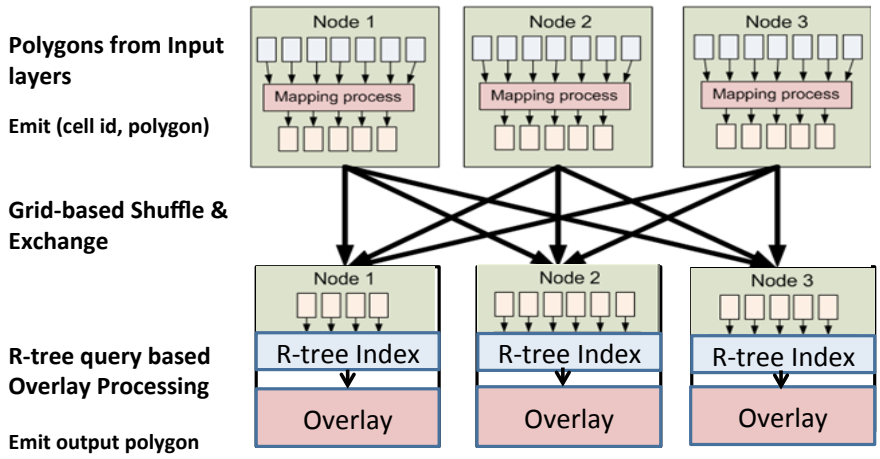

Figure (2.1) *Polygon Overlay in Hadoop Topology Suite (HTS) using a single Map and Reduce phase.*

2.4.2 Spatial Overlay and Join using Hadoop

Parallelizing polygonal overlay and join with MapReduce has its own set of challenges. MapReduce focuses mainly on processing homogeneous data sets, while polygon overlay and join are binary-input problems that has to deal with two data sets. Moreover, partitioning vector data evenly to yield good load balance is non-trivial.

A suite of overlay algorithms, known as *Hadoop Topology Suite (HTS)*, is presented in [16, 17] by employing MapReduce in three different forms, namely, i) with a single map and reduce phase, ii) with chaining of MapReduce jobs, and iii) with a single map phase only. *HTS* is based on R-tree data structure which works well with non-uniform data. *HTS* carries out end-to-end overlay computation starting from two input GML files, including their parsing, employing the bounding boxes of potentially overlapping polygons to determine

the basic overlay tasks, partitioning the tasks among processes, and melding the resulting polygons to produce the output GML file.

The basic idea is to perform filtering step in map phase and perform polygon clipping in the reduce phase. This works because the geometric computations are performed on large sets of independent spatial data objects and thus, naturally lend themselves to parallel processing using map and reduce paradigm. Initially, the dimension of the grid is determined which is a minimum bounding box spatially containing all the polygons from base layer and overlay layer. Then, the dimension of grid cells is computed based on the number of partitions. The number of cells is greater than the reduce capacity of the cluster in order to ensure proper load-balancing. A polygon may belong to one or more cells and since the bounding boxes of all the grid cells are known, each mapper task can independently determine to which cell(s) a polygon belongs to.

Figure 2 shows the *HTS* architecture using one map and one reduce phase. First, each mapper process reads the polygon and parses the MBR and polygon vertices. Then, the polygons are emitted as (key, value) pairs, where the cell ID is the key and polygon vertex list is the value. After all the mappers are done, each reducer gets a subset of base layer and overlay layer polygons corresponding to a grid cell. In each reducer, a local R-tree is built from the overlay layer polygons and for every base layer polygon, the intersecting overlay layer polygons are found out by querying the R-tree. Finally, the overlay operation is carried out using polygon clipping library and output polygons are written to HDFS.

Hadoop-GIS [18] is a scalable and high performance spatial data warehousing system on Hadoop and it supports large scale spatial join queries. According to the authors, its performance is on par with parallel SDBMS and outperforms SDBMS for compute-intensive queries. It is available as a set of library for processing spatial queries, and as an integrated software package. The evaluation of the system is done using pathology imaging data and OpenStreetMap data. *SpatialHadoop* [19, 20] is an open source MapReduce framework for large spatial data that supports spatial joins. In addition to spatial join, *SpatialHadoop* is shipped with built-in spatial high level language, spatial data types, spatial indexes, and effi-

cient spatial operations. Both *Hadoop-GIS* and *SpatialHadoop* used spatial indexing whereas Zhang et al. [21] used sweepline algorithm for spatial join.

Authors in [9] have taken a parallel database approach for spatial join. The unique features of their system known as *Niharika* are spatial declustering and dynamic load balancing on top of a cluster of worker nodes, each of which runs a standard PostgreSQL/PostGIS relational database. In [22] Ray et al. discussed a skew-resistant parallel in-memory spatial join using Quadtree declustering, which partitions the spatial dataset such that the amount of computation demanded by each partition is equalized and the processing skew is minimized. *SpatialHadoop* and *Niharika* have been shown to run on Amazon EC2 cluster [9, 23].

2.4.3 Spatial Overlay and Join using GPUs

GPUs have been successfully utilized in many scientific applications that require high performance computing. Architecturally, a CPU is composed of only a few cores that can handle a limited number of software threads at a time. In contrast, a GPU is composed of hundreds to thousands of cores that can handle thousands of threads simultaneously. In addition to providing a very high level of thread parallelism, the GPUs coupled with high throughput memory architectures achieve significant computational throughput. As a result, leveraging the parallel processing power of GPUs for speeding up overlay and join computation has been explored recently. However, the architecture of GPUs is fundamentally different than CPUs; thus, traditional computational geometry algorithms cannot simply be run on a GPU. GPUs exhibit Single Instruction Multiple Data (SIMD) architecture and many algorithms, including plane-sweep algorithms, do not naturally map to SIMD parallelism. As a result, new algorithms and implementations need to be developed to get performance gains from GPU hardware.

GPU-accelerated parallel rectangle intersection algorithms are discussed in [24, 25]. In [24], all intersecting pairs of iso-oriented rectangles are reported using GPU. The result shows over 30x speedup using Geforce GTX 480 GPU against well implemented sequential algorithms on Intel i7 CPU. In [26], Wang et al. used hybrid CPU-GPU approach to find

intersection and union of polygons in pixel format. McKenney et al. [27] developed GPU implementation of line segment intersection and the arrangement problem for overlay computation. For some datasets, authors show that their implementation of geospatial overlay on Nvidia Tesla GPU runs faster than plane-sweep implementation. In [28], Franklin's uniform grid based overlay algorithm [29] is implemented using OpenMP in CPUs and CUDA on GPUs. The robustness issue is handled by snap rounding technique and improved performance is achieved by computing the winding number of overlay faces on the uniform grid efficiently using CPUs as well as GPUs. At first, edges of spatial features are mapped to grid cells. According to the paper, 10 to 20 edges per cell on average minimizes execution time. The Single Instruction Multiple Thread (SIMT) architecture of a GPU is used to detect intersecting edges by all-to-all comparison of edges. Simple point-in-polygon tests are executed in parallel to find the output vertices. Experimental results using real-world datasets showed their multi-threaded implementation exhibiting strong scaling with an efficiency of 0.9 on average. GPU implementation took 4 times less time when compared with CPU version. A GPU implementation of Greiner-Hormann polygon clipping algorithm [30] is discussed in [8].

Authors in [31] describe how a spatial join operation with R-Tree can be implemented on a GPU. The algorithm begins by parallel filtering of objects on the GPU. The steps of the algorithm are as follows:

1. The data arrays required for the R-Tree are mapped to the GPU memory.

2. A function to find an overlap between two MBR objects is executed by threads on the GPU in parallel.

3. The set of MBRs from Step 2 are checked whether they are in the leaf nodes or not. If they are in the leaf nodes, return the set as the result and send them to the CPU. If they are not the leaf nodes, then they are used as input again recursively until reaching leaf nodes.

Other recent work on GPU based parallel spatial join is discussed in [32–34].

2.4.4 Accelerated Hadoop for Spatial Join

While MapReduce can effectively address data-intensive aspects of geospatial problems, it is not well suited to handle compute-intensive aspect of spatial overlays and joins. A high performance approach can leverage parallelism at multiple levels for these compute-intensive operations. Most spatial algorithms are designed for executing on CPU, and the branch intensive nature of CPU based algorithms require them to be redesigned for running efficiently on GPUs. Haggis [35] first transforms the vector based geometry representation into raster representation using a pixelization method. The pixelization method reduces the geometry calculation problem into simple pixel position checking problem, which is well suited for executing on GPUs. Since testing the position of one pixel is independent of another, the computation is be parallelized by having multiple threads processing the pixels in parallel.

2.5 Future Directions

Nowadays, a single machine may be equipped with multiple processors, e.g., multi-core CPUs and GPUs. Such hardware configurations are available on major cloud computing platforms and supercomputers. Moreover, in the coming years, such heterogeneous parallel architecture will become dominant, and software systems must fully exploit this heterogeneity to deliver performance growth [36]. Cluster of computing nodes with accelerators including GPU and Many Integrated Core (MIC) have been already utilized for GIS computations [37, 38]. Hybrid architectures, employing for example, MPI with GPU acceleration and Hadoop with GPU acceleration may be promising for many compute-intensive as well as data-intensive geospatial problems. A recent vision article [39] projects that the bulk of geo-spatial software packages will employ both distributed and shared memory parallel processing including the GPUs toward scalable solutions.

PART 3

OUTPUT-SENSITIVE PARALLEL ALGORITHM FOR POLYGON CLIPPING

3.1 Introduction

Polygon clipping is one of the complex operations in computational geometry. It is a primitive operation in many fields such as Geographic Information Systems (GIS), Computer Graphics and VLSI CAD. Sequential algorithms for this problem are in abundance in literature but there are very few parallel algorithms solving it in its most general form. We present the first output-sensitive CREW PRAM algorithm, which can perform polygon clipping in $O(\log n)$ time using $(n + k + k')$ processors, where n is the number of vertices, k is the number of edge intersections and k' is the additional temporary vertices introduced due to the partitioning of polygons. The current best algorithm by Karinthi, Srinivas, and Almasi [40] does not handle self-intersecting polygons, is not output-sensitive and must employ $\Theta(n^2)$ processors to achieve $O(\log n)$ time. Our algorithm is developed from the first principles and it is superior to [40] in cost. It yields a practical implementation on multicores and demonstrates 30x speedup for real-world dataset. Our algorithm can perform the typical clipping operations including intersection, union, and difference.

In geometry, a polygon is traditionally a plane figure that is bounded by a finite chain of straight line segments closing in a loop to form a closed chain. A polygon can be classified as a *simple* polygon in case its line segments do not intersect among themselves, otherwise it is termed as *self-intersecting*. A polygon is *convex* if every interior angle is less than or equal to 180 degrees otherwise it is *concave*. Geometric intersection problems arise naturally in a number of applications. Examples include clipping in graphics, wire and component layout in VLSI and map overlay in geographic information systems (GIS). Clipping polygons is a fundamental operation in image synthesis and it is widely used in Computer Graphics.

While clipping usually involves finding the intersections (regions of overlap) of subject and clip polygons, some clipping algorithms can also find union and difference. Clipping an arbitrary polygon against an arbitrary polygon is a complex task. The major computation in clipping involves 1) finding edge intersections, 2) classifying input vertices as contributing or non-contributing to output, and 3) stitching together the vertices generated in Step 1 and 2 to produce one or more output polygons.

The polygonal data has high degree of irregularity. For two polygons with n and m edges, the number of intersections can be $O(nm)$. Plane-sweep based polygon clipping algorithm has time complexity of $(n + k)\log n$ where k is the number of intersections [3]. Convex polygons are easier to handle in comparison to non-convex and self-intersecting polygons. With polygons that are not self-intersecting, it suffices to test edges taken from two different polygons for intersection detection. On the other hand, self-intersecting polygons increases the number of edge pairs for intersection detection since now an edge from a polygon can intersect with other edges from the same polygon. In general, the complexity of clipping operation between two polygons depends on 1) number of input polygon vertices, 2) number of intersections, and 3) self-intersection in edges. GIS polygonal datasets may contain large number of arbitrary polygons. Some of the U.S. State boundaries consists of about 100,000 polygonal edges. Parallel polygon clipping algorithms in these scenarios can improve the performance when compared with sequential clipping algorithms.

In this paper, we present an output-sensitive algorithm for the parallelization of Vatti's polygon clipping algorithm [2] using PRAM model. Parallel output-sensitive algorithms are those whose complexity bounds depend on not only the size of the input but also the size of the output. For example, when computing the intersections of n edges, the time and number of processors used by an algorithm may depend on n as well as the number of the actual intersections between the given edges.

The specific technical contributions are as follows:

1. We present parallelization of a plane-sweep based Vatti's algorithm relying only on primitives such as prefix sum and sorting, and handling arbitrary polygons. The in-

sights and algorithmic tools developed for the plane-sweep based algorithm may also be useful elsewhere.

2. This algorithm is the first output-sensitive polygon clipping algorithm with $O(((n + k + k')\log(n + k + k'))/p)$ time complexity using p processors, where $k' \leq O(n^2)$ is the additional temporary vertices introduced due to the partitioning of polygons and $p \leq O(n+k+k')$. This improves the current state of art which is not output-sensitive, always employs $\theta(n^2)$ processors and does not handle self-intersecting polygons [40].

   We present a new technique to parallelize plane-sweep based on 1) counting the pairs of inversions and 2) reporting them. This technique can be easily implemented by parallel sorting as opposed to some of the existing techniques that use complex data structures such as array-of-trees or plane-sweep trees [41,42] to parallelize plane-sweep.

3. We also present a practical, multi-threaded version of the plane-sweep based algorithm and its implementation for clipping of arbitrary polygons, and validate its performance using real-world and simulated datasets. This resulted in 30x speedup using a 64-core 1.4 GHz AMD Opteron Processor for real-world dataset containing large number of polygons when compared to sequential clipping libraries and ArcGIS which is the fastest currently available commercial software application. For baseline performance, ArcGIS version 10 running on 64-bit Windows 7 with AMD Phenom Processor 2.7 GHz with 8 GB memory was used.

The rest of this paper is organized as follows: Section 6.2 reviews the relevant literature briefly, introduces various operations that define polygon clipping, and reviews the segment tree [43] data structure utilized in our algorithm. Section 3.3 describes our plane-sweep based polygon clipping algorithm, with rigorous design and time complexity analysis. Our multi-threaded algorithm is in Section 3.4 and its experimental results are in Section 6.4.

3.2 Background and Literature

3.2.1 The Computational Model

The PRAM (parallel random access machine) model of parallel computation is a fundamental shared-memory model where the processors operate synchronously. The CREW (concurrent read exclusive write) version of this model allows many processors to simultaneously read the content of a memory location, but forbids any two processors from simultaneously attempting to write in the same memory location. Since we wish to solve polygon clipping in an output-sensitive manner, we allow the number of processors to grow based on the output size. As such, additional processors are requested (if necessary) constant number of times. This previously-established approach has been employed, for example, in [41, 44], and [45], for line segment intersection problems. We note that [45] also employed dynamic spawning, which we do not need.

3.2.2 Polygon clipping

Sutherland-Hodgman [46] and Liang-Barsky clipping [47] algorithms are sequential clipping algorithms which do not work for arbitrary polygons and the existing parallel clipping algorithms we found in literature are parallelization of these sequential algorithms with implementation done on classic parallel architectures [48–51]. Vatti's algorithm is sequential polygon clipping algorithm which takes arbitrary polygons as input. To our best knowledge, there is no parallelization of this algorithm in literature. Apart from Vatti's algorithm, plane-sweep based sequential polygon intersection algorithms are also discussed in [3, 52, 53]. Parallelization of plane-sweep algorithm for multi-cores is discussed in [54, 55]. However, our parallelization focuses on polygon clipping specifically and our partitioning and merging techniques are also different from [54]. The algorithm described in [55] is applicable for rectangles.

Limitations of Current Parallel Algorithms: There are only a few papers on computing set operations on polygons in parallel [40, 56] (even though there are several parallel

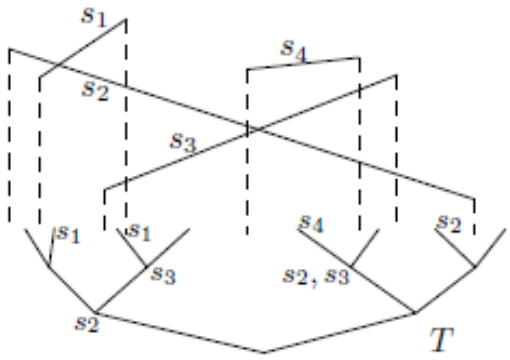

Figure (3.1) *Segment Tree*

algorithms for line intersections [41, 44, 45]). Self-intersections are not handled in [40]. The time complexity of algorithm described in [40] is $O(\log n)$ using $\theta(n^2)$ processors on EREW PRAM model, thus is not sensitive to output size. Other solutions for parallel polygon clipping [48–51] are limited to certain types of polygons and not output sensitive. A uniform grid based partitioning approach is discussed in [56] where a grid is superimposed over the input polygons. Intersection finding operation and output vertex processing is performed in parallel in the grid cells. Thus, this works well only with good load distribution. A survey on parallel geometric intersection problems is presented in [57]. In contrast, our parallel algorithm is output-sensitive since our algorithm does not need to know the number of edges in the output polygon(s) in advance. It also does not need to know the number of processors required to optimally solve the problem in advance. Moreover, it can handle arbitrary polygons.

3.2.3 Segment Tree

We employ segment tree data structure. It is a complete binary tree as shown in Figure 3.1. Its leaves correspond to the elementary intervals induced by the endpoints of segments and its internal nodes correspond to intervals that are the union of elementary intervals. The internal nodes contain list of segments (cover list) which span the node's

range (i.e., the node range is a subinterval of the horizontal projection of these segments), but do not span the range of node's parent. A query for a segment tree, receives a point p, and retrieves a list of all the segments stored which contain p in $O(\log n + k)$ time where k is the number of retrieved segments. A segment tree with cover lists can be constructed in $O(\log n)$ time using $O(n)$ processors on CREW PRAM model [58]. Variations of segment tree known as plane-sweep tree has been explored in literature to parallelize plane-sweep algorithm for line-segment intersection problem [41, 45].

3.3 Algorithm for clipping two polygons

3.3.1 Terminology

We present the clipping algorithm for two input polygons namely subject polygon and clip polygon. This algorithm can be extended to handle two sets of input polygons. A polygon can be viewed as a set of left and right bounds with respect to the interior of the polygon as described in [2]. A *bound* comprises of edges starting at a local minima and ending at a local maxima. All the edges on the left *bound* are called $left$ edges and those on the right are called $right$ edges. Similarly, vertices can be labeled as *minima*, *maxima*, $left$ and $right$ [2]. When two polygons are overlaid to produce the output polygon, some edges/vertices will be part of the output and we call them as *contributing* edges/vertices. An imaginary horizontal line called *scanline* (also known as sweepline) is drawn through each vertex such that there are no vertices in between two such lines. The area between two successive scanlines form a *scanbeam*.

3.3.2 Vatti's Polygon Clipping Algorithm

Vatti's algorithm uses a variation of plane-sweep method to clip a pair of polygons or two sets of polygons. Start and end vertices of edges are treated as event points. The algorithm creates minima table to store the minima list and bounds of subject and clip polygon. Also, a scanbeam table is created to store the event point schedule. An active edge list E containing all the edges passing through a scanbeam is maintained and updated

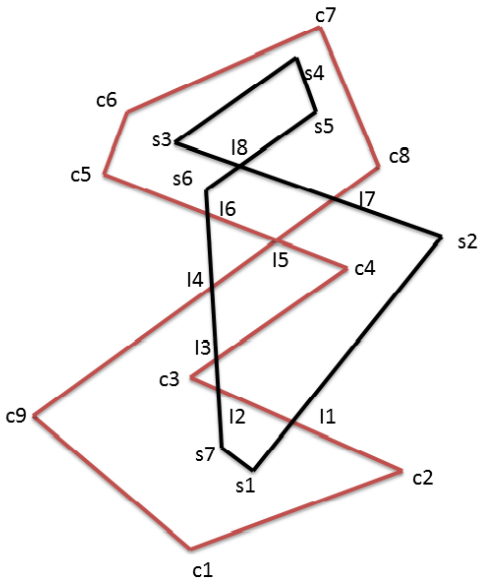

Figure (3.2) *Example showing intersection of two self-intersecting concave polygons.*

dynamically as the plane-sweep progresses from one scanbeam to another. The gist of Vatti's algorithm is to scan every vertex in the input polygons from bottom to top and compute intersection points in E. The output polygon is constructed incrementally by stitching the contributing vertices and intersections together based on their labels.

The intersection of two polygons with self-intersecting edges is shown in Figure 4.1. Let us consider scanbeam $s2-s6$. The active edges passing through this scanbeam are $s7s6$, $c9c8$, $c4c5$ and $s2s3$. At first, intersecting vertices $I6$ and $I7$ are found out and processed. When two polygonal edges intersect to generate a vertex, then the nature of the edge intersection can determine the relative position of the vertex in the output polygon. The vertex labeling rules are discussed in Vatti's algorithm [2]. For example, $I6$ is the intersection between edges $c4c5$ and $s7s6$, both with label $left$, $I6$ is tagged with $left$ label. Applying the vertex labeling rules, $I2$ is tagged with *maxima* label and $I3$ is tagged with *minima* label. A *minima* signifies initialization of new polygon and a *maxima* signifies termination of a polygon. Moreover, these labels also determine the relative position of the vertices in the output polygon. We use these insights in our parallel algorithm.

In subsection 3.3.5, we show how active edges can be independently determined in the scanbeams by concurrently querying a parallel segment tree. Once the active edges are determined, these edges can be labeled independently as shown in *Lemma* 1 and contributing vertices can be identified independently as shown in *Lemma* 2 in all the scanbeams.

3.3.3 Problem Definition

Let polygon $B = (V_b, E_b)$ where $V_b = \{b_1, b_2, .., b_n\}$, $b_i \in B$ represent n vertices and $E_b = \{\overline{b_1b_2}, \overline{b_2b_3}, .., \overline{b_{n-1}b_n}, \overline{b_nb_1}\}$ represent the edges. Let polygon $O = (V_o, E_o)$ where $V_o = \{o_1, o_2, .., o_t\}$, $o_i \in O$ represent t vertices and $E_o = \{\overline{o_1o_2}, \overline{o_2o_3}, .., \overline{o_{t-1}o_t}, \overline{o_to_1}\}$ represent the edges. Each vertex is represented by a pair of cartesian coordinates x and y. Let I represent vertices of intersection among edges in B and O. For simplicity, we assume that there are no horizontal edges (parallel to x-axis) present. If horizontal edges are present then we assume that the edges are preprocessed by slightly perturbing (less than pixel width) the vertices to make them non-horizontal. Vatti's algorithm handles this scenario as a special case. Given, B, O and an operator *op*, we want to find the set P, which represents the new polygon(s) consisting of vertices from V_b, O_b and I and edges from E_b, E_o and new edges induced by I. The set P may represent zero or more polygons (zero in case when input polygons do not overlap). Clipping may produce a point or a line. For simplicity, we assume no such cases. We concentrate on output which can be a convex polygon or a concave polygon or a set of convex and/or concave polygons.

3.3.4 Multi-Way Divide and Conquer Algorithm

Overview of plane-sweep based algorithm: Let Y = $\{y_1, y_2, .., y_m\}$ be an ordered set of vertices formed by projecting the edge start and end vertices on y-axis. Each of the m-1 scanbeams from bottom to top are associated with intervals $\{(y1, y2), (y2, y3), .., (y_{m-1}, y_m)\}$. The intervals $\{y_i, y_{i+1}\}$ with y_i equal to y_{i+1} are not considered as they do not form a valid scanbeam.

The edges in polygons B and O are merged together and partitioned into multiple

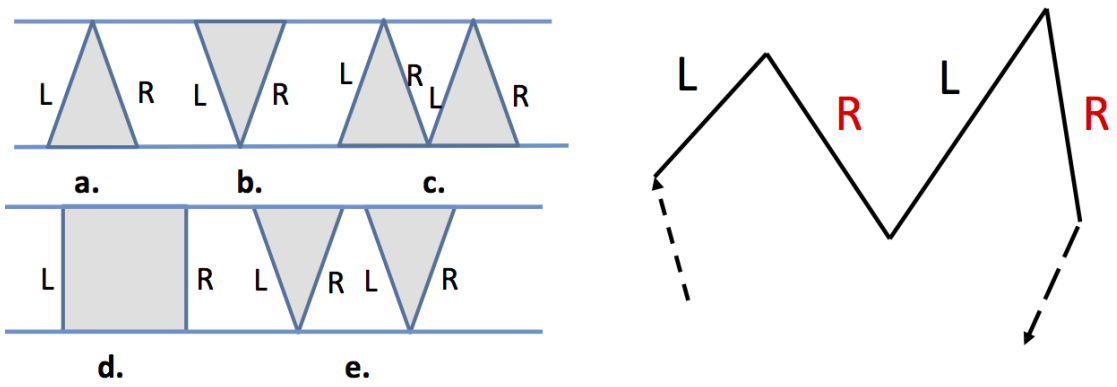

Figure (3.3) *(I) Different types of partial polygons in a scanbeam with L/R edge labels (L stands for left, R stands for right), (II) Assigning labels to edges in a scanbeam*

scanbeams. Partitioning the input polygons by scanlines decomposes the original problem into subproblems (of various sizes) represented by the scanbeams, suitable to apply divide and conquer approach. New geometrical figures (*partial polygons*) are formed by intersecting the m scanlines with B and likewise with O. The additional vertices thus produced by intersecting scanlines with input polygons are denoted as *virtual* vertices and these are stored along with individual partial polygons as *pivots* which can be used later in the merging phase. Figure 3.3(I) shows examples of partial polygons that are possible in a scanbeam. Also note in this figure, how the *left* (L) and *right* (R) edge labels alternate one another in a scanbeam. In *Lemma* 1, we provide a proof sketch for this property.

The following results are on concurrent processing of vertices and edges in a scanbeam and its time complexity.

Lemma 1: Labeling Locally: The polygon edges in a scanbeam can be labeled as *left* or *right* without looking at the overall geometry and the labels alternate one another (Figure 3.3).

Proof sketch: Based on the property of winding number, it is shown that an imaginary ray that intersects the polygon exactly once traverses either from the interior to the exterior of the polygon, or vice versa [59]. Based on this fact, we can observe that if a point moves along a ray from infinity and if it crosses the boundary of a polygon then it alternately goes

from the outside to inside, then from the inside to the outside. Since, the edge labeling is defined in terms of interior of a polygon, if edges are arranged in the order in which they intersect the scanline, the edges of a polygon can be labeled as *left* and *right* alternately. □

Thus, from *Lemma 1*, we can derive an algorithm to find the label of an edge based on its index in the sorted edge list in a scanbeam. An intersection vertex can be labeled as $Left$, $Right$, $Minima$ or $Maxima$ based on the labels of the intersecting edges using Vatti's vertex labeling rules [2]. Thus, as a consequence of *Lemma* 1, relative position of a vertex can also be independently determined in the partial output polygon.

Lemma 2: Independent Identification of Contributing Vertices: A vertex can be classified as contributing or not independently in a scanbeam.

$Proof sketch$: In a scanbeam, a vertex can either be a start/end vertex or a new intersection. As such, two cases are possible:

Case 1: Intersection vertex: The vertices lying in between the scanlines are intersecting points which can be either i) intersections between the edges from the same polygon or ii) from different polygons. In case ii), the vertices are contributing irrespective of the clipping operation involved. Case i) depends on the clipping operation involved and it can be resolved using *point-in-partial-polygon test* using only the edges in a scanbeam which we explain shortly.

Case 2: Segment start or end vertex: A start/end vertex may lie either on the top or bottom scanline. In both cases, one can decide whether this vertex is contributing or not depending on its position with respect to other vertices on the scanline using *point-in-partial-polygon* test. □

Point-in-partial-polygon test: In a sequential point-in-polygon testing, one pass through all the edges of a polygon is required in order to determine if the point is contributing or not. But, when scanbeams are already created, this test can be performed faster since only the edges in a given scanbeam needs to be considered, other edges can be safely ignored. Assuming that the edges in a scanbeam denoted by E are sorted (by x-coordinate of edge's intersection with scanline), in order to find if a vertex from subject polygon is contributing

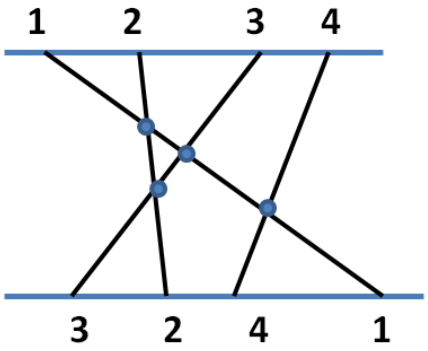

Figure (3.4) *Intersection of 4 edges in a scanbeam. The inversion pairs are (3,1), (3,2), (4,1), (2,1) for the list {3,2,4,1}. These pairs also represent the intersecting edges.*

or not, the number of edges from clip polygon to its left in E are counted. If the count is odd then the edge is contributing otherwise non-contributing. This parity test can be expressed as a standard prefix sum problem.

Lemma 3: Contributing Vertices in $O(\log n)$ ***Time***: On PRAM model, a logarithmic time algorithm can be developed to find whether a vertex on a scanline is contributing or not.

$Proof sketch$: Let us consider a set E of n edges in a scanbeam which is sorted on the x-coordinate of intersection of E with a given scanline. For this ordered set, let $L = [l_1, l_2, .., l_n]$ represent labels for the corresponding edges in E and a binary associative operator $+$. Assign 0 to the label of the edges belonging to B and 1 to those belonging to O. A prefix sum for L is denoted by $P\ [l_1, (l_1 + l_2), .., (l_1 + l_2 + .. + l_n)]$, where P_i is represented as $(l_1 + l_2 + .. + l_i)$. A vertex v of an edge $E_i \in B$ lying on a scanline, is contributing if and only if P_i is odd. Now, repeat the prefix sum by reversing the 0/1 label for the edges of B and O to determine the contributing vertices for polygon O. This all-prefix-sum even-odd parity test can find all the contributing vertices in a scanbeam by invoking it twice (for lower and upper scanline). This algorithm requires sorting and prefix sum computation. With n processors, both operations can be done in $O(\log n)$ time [60]. □

Finding Intersections using Inversions: If the edges span a bounded region,

number of edge intersections can be found out within the region simply by knowing the order in which the edges intersect the boundary of the region [61]. For example, as shown in Figure 3.4, the order of edges L intersecting the lower scanline is $\{3, 2, 4, 1\}$ and the number of inversions in L is equal to the number of edge intersections in the scanbeam. In the worst case, there can be $O(n^2)$ inversions in an unsorted list but by extending mergesort, inversions in a scanbeam can be counted in $O(n\log n)$ time sequentially. If we add the inversions for all the scanbeams, the number of intersections between two arbitrary polygons can be found out. To allocate processors in an output-sensitive manner, we need to first find the number of intersections and then allocate that number of additional processors to the pairs of intersecting segments.

Lemma 4: The number of intersections in a scanbeam can be computed in $O(\log n)$ time using $O(n)$ processors.

Proofsketch: We show how parallel mergesort can be extended to find the intersecting pairs of edges by counting the number of inversions first and reporting them subsequently. Cole's mergesort [62] is a pipelined algorithm which works at several levels of the tree at once and overlaps the merging process at different nodes. Let us consider left sublist $A_l = \{A_1, .., A_{mid}\}$ and right sublist $A_r = \{A_{mid+1}, .., A_n\}$ of edge indices present in two children of an internal node of the binary merge tree whose leaf nodes contain the edge indices. The inversions in the leaf nodes have to be identified and counted while sorting A_l and A_r. Let the number of inversions found in A_l and A_r be Inv_l and Inv_r respectively. To illustrate merging, let us consider that in a given timestep while merging the two sublists, we are at index i in list A_l and at index j in list A_r. Since, A_l and A_r are already sorted, if at any step, $A_l[i] > A_r[j]$, then $\{A_l[i+1], .., A_l[mid]\}$ will also be greater than $A_r[j]$. As such Inv_m for an element at j consists of set $\{(i, j), (i + 1, j), .., (mid, j)\}$ containing $|mid - i + 1|$ inversions. Inv_l and Inv_r are added to the number of inversions found during the merging (Inv_m) of sorted list A_l and A_r for each internal node of the binary tree. With this modification, the total number of inversions (Inv_{root}) is available in the root node of the merging tree when mergesort algorithm terminates. □

Table (3.1) *Example shows the merging of sublists A_l and A_r in an internal node of Cole's mergesort tree in a time-stepped fashion. A_l = {5,6,7,9} and A_r = {1,2,3,4}. Also note the inversions marked for reporting by our extended Cole's merging algorithm.*

Time Step(i)	**Comparison ($A_l(i)$, $A_r(i)$)**	**Merged List**	**Inversions**
3	$7, 3$	$1, 2, 3, 4, 5, 6, 7, 9$	(7,1), (7,2), (7,4) (7,3), (5,3), (6,3), (9,3)
3	$5, 1$	$1, 2, 4, 5, 6, 9$	(5,1), (5,2), (5,4) (6,1), (9,1)
2	$6, 2$	$2, 4, 6, 9$	(6,2), (6,4), (9,2)
1	$9, 4$	$4, 9$	(9,4)

In case of Cole's algorithm, the left (A_l) and right sublists (A_r) are dynamically growing and getting merged together according to the timestep as shown in Table 3.1. At first timestep, every fourth element from A_l and A_r is compared. In the second timestep, similarly, every second element is compared. In the last timestep, remaining elements are compared. At each timestep, inversions are marked by utilizing the cross-rank of the elements computed by the original Cole's algorithm in left and right sublists. Please refer to [62] for details on Cole's algorithm.

Identifying pairs of intersecting segments: As described above, in the extended Cole's mergesort algorithm, each non-leaf node has to update Inv_l, Inv_r, Inv_m and add them together to generate Inv for its parent. Based on the total count K of inversions found across all the scanbeams, K additional processors are allocated which can be $O(n^2)$. These are subsequently allocated proportionately to individual scanbeams (Inv_{root}) and to the individual mergetree nodes for each of the $O(\log n)$ merging steps of (i,j) index pairs which have non-zero inversions based on their respective inversion counts. The merging process is repeated to enable recording of the inversions. In the process of merging, for each

set of inversions found for the jth element from A_r, the processors assigned to the node (as in Cole's original algorithm) are utilized to copy the elements from location i to mid from A_l and j to two different arrays I and J respectively. Two additional auxiliary arrays Cnt and Sum are also obtained from the initial mergesort to aid this process. Cnt stores the $(mid - i + 1)$ values (for $O(\log n)$ time steps, thus of total size $O(n\log n)$) and Sum stores the prefix sum of Cnt. This computation for inversion counting and prefix sum can both be accommodated in the original run of the mergesort algorithm without sacrificing its $O(\log n)$ time complexity using $O(n)$ processors.

Using Cnt and Sum arrays, a processor reads one element from I and the other from J and in this way, Inv_{root} number of processors can report all the inversions, which represents indices of the intersecting pair of edges in a scanbeam, in constant time.

Plane-Sweep Based Algorithm: With these results, we discuss the parallelization of Vatti's clipping algorithm. The parallel algorithm consists of four steps and is given next as Algorithm 1. The scanbeams are generated by Step 1. Step 2 is meant for populating the scanbeams with edges from input polygons in parallel. In subsection 3.3.5, we show the application of segment trees to accomplish the partitioning task in logarithmic time.

Algorithm 1 Plane-Sweep based Divide-and-Conquer Algorithm

INPUT: $V = V_b \cup V_o$, $E = E_b \cup E_o$ and clipping operator $op \in \{\cap, \cup, \setminus\}$
OUTPUT: $P \leftarrow B\ op\ O$
Step 1: Sort V by the y-coordinates.
Step 2: Partition E into m subsets E_1, E_2, ... , E_m of edges where E_i belongs to ith scanbeam and m is the number of scanbeams.
Step 3:
for all E_i in E **in parallel do**
  Get y-coordinate of lower(y_b) and upper(y_t) scanline
  Step 3.1: $P_i \leftarrow$ Initialize(y_b) // *process minima*
  Step 3.2: $P_i \leftarrow P_i \cup$ EdgeIntersection(y_b, y_t)
  Step 3.3: $P_i \leftarrow P_i \cup$ Terminate(y_t) //*process maxima*
  Step 3.4: Arrange the vertices in P_i in correct order
end for
Step 4: $P \leftarrow (P_1 \cup P_2 \cup ... \cup P_m)$

Table (3.2) *Scanbeam table showing the edges and labeled vertices of output polygons after processing intersections, start/end points for all scanbeams.*

Scanbeam	**Edges**	**Partial Polygon**
$s_1 - s_7$	$c_1c_9, s_1s_7, s_1s_2, c_2c_3$	$s_1^S, \quad s_7^L$
$s_7 - c_9$	$c_1c_9, s_7s_6, s_1s_2, c_2c_3$	i_1^R
$c_9 - c_3$	$c_9c_8, s_7s_6, s_1s_2, c_2c_3$	i_2^T
$c_3 - c_4$	$c_9c_8, c_3c_4, s_7s_6, s_1s_2$	$i_4^L, \quad i_3^S, \quad c_4^R$
$s_2 - s_6$	$s_7s_6, c_9c_8, c_4c_5, s_2s_3$	$s_6^L, \quad i_6^R, \quad i_5^L, \quad i_5^R, \quad i_7^R$
$c_5 - s_3$	$c_5c_6, s_6s_5, s_2s_3, c_8c_7$	$s_3^L, \quad i_8^L, \quad i_8^R$
$s3 - s5$	$c_5c_6, s_3s_4, s_6s_5, c_8c_7$	s_5^R
$c_6 - s_4$	$c_6c_7, s_3s_4, s_5s_4, c_8c_7$	s_4^T

Here, partitioning of polygon edges by m scanlines leads to decomposition of the input polygons into trapezoids, triangles, and self-intersecting figures. In case the polygon is simple, a vertex in partial polygon is either an endpoint of an edge or a point where a scanline crosses an edge. Due to the horizontal scanlines, no corner that is an edge starting-point or ending-point can have an interior angle greater than 180 degrees. Moreover, any angle at a point where a scanline crosses an edge must be less than or equal to 180 degree as well. The horizontal scanlines have removed all non-convexities (see Figure 3.3 and 3.6) as such the new geometrical figures thus formed are convex. We stress on the convexity of the partial polygons here since it simplifies the formation of *bounds* for partial output polygon. Non-simple partial polygons in a scanbeam with self-intersections are handled as a special case. Each scanbeam can be processed concurrently to yield partial output polygon which can be finally melded together into one global solution. The merge operation is expressed as $P \leftarrow (P_1 \cup P_2)$.

Each set E_i is present as two sorted sequences E_L and E_U, the elements of E_L are sorted according to x-coordinate of their intersections with lower scanline and elements of

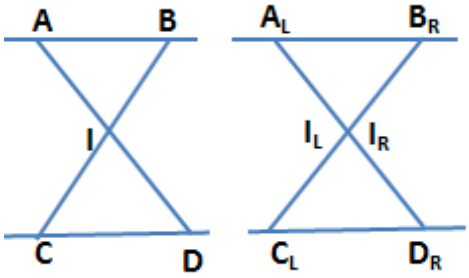

Figure (3.5) *Labeling of self-intersection I as left and right.*

E_U according to that with the upper scanline. The edges in E_i are labeled according to *Lemma* 1. In Step 3.2, inversion finding and reporting steps are performed as discussed in *Lemma* 4 to find intersections. Based on the edge labels, contributing vertices are found in Step 3 based on *Lemma* 2 and 3. Table 3.2 shows the output generated by the parallel polygon clipping algorithm for the polygons as shown in Figure 4.1. For each output vertex, the superscript shows the labels which are S (start), T (terminate), L (left) and R (right).

Merging labeled vertices in a scanbeam: The input to Step 3.4 is a list of labeled vertices in P_i which needs to be arranged in order to generate partial output polygon(s). The partial output polygons formed by clipping operations on trapezoids in a scanbeam may be convex or concave polygons. The *intersection* operation results in convex output since the trapezoids are themselves convex in nature. On the other hand *union* operation may result in concave output.

For the intersection operation, the vertices with *left* and *right* labels are kept in two subgroups (*bounds*). The vertices with *left* label are sorted in ascending order by y-coordinate to form $left$ *bound* and the vertices with *right* label are sorted in descending order by y-coordinate to form *right bound*. Then, the $left$ and *right* bounds are concatenated to yield a partial polygon P_i based on the parity of the scanbeam id. For scanbeams with even parity, *right* bound is followed by $left$ bound and for scanbeams with odd parity, $left$ bound is followed by *right* bound. Figure 3.3 and 3.5 shows different scenarios of processing the labeled vertices in a scanbeam. There may be single partial polygon as shown in Figure 3.3

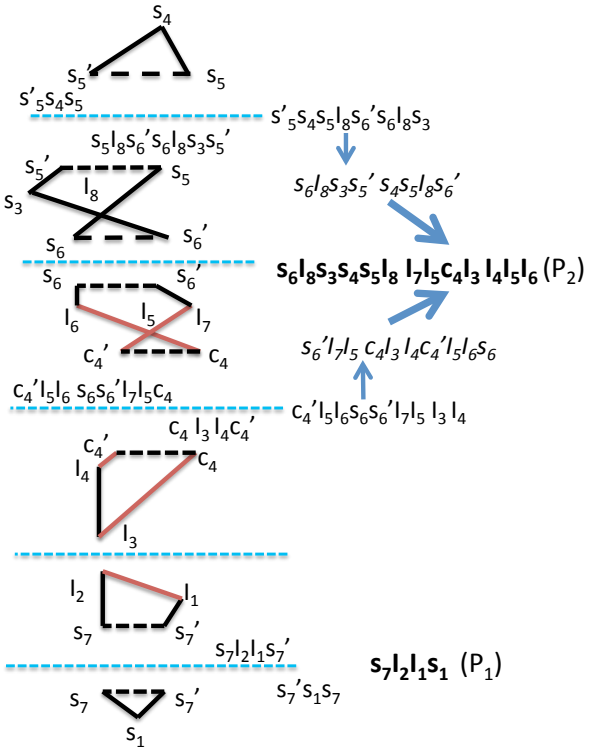

Figure (3.6) *Merging partial output polygons from scanbeams. The arrows show the order of merging.* P_1 *and* P_2 *are the output polygons*

(a, b and d) or multiple partial polygons as in Figure 3.3 (c and e) in a scanbeam. When there is self-intersection I as shown in Figure 3.5 in a scanbeam, it is duplicated and labeled as $left$ and $right$. The partial polygon in this scenario is $(D_R, I_R, B_R, A_L, I_L, C_L)$. As we can see, the local merging is performed simply by concatenating *bounds* since the input partial polygons are convex in nature. Similarly, for other clipping operations like *union* and *difference*, local merging can be done using simple concatenation of *bounds* as described in [63].

Merging partial output polygons: We partitioned the input polygons by introducing extra *virtual* vertices which makes merging by union operation straightforward since partial polygons P_i and P_{i+1} from two adjacent scanbeams may have a common horizontal edge or a vertex. If the partial polygons do not share a common edge/vertex, then union operation

simply adds the partial polygon to the list of output polygons. Otherwise, the union operation can be implemented by concatenating the vertex list of polygons in a certain order as shown in Figure 3.6. Partial polygons in two consecutive scanbeams are concatenated together at a time and subsequently rearranged for merging in the next phase. The merging is described as m-leaf complete binary tree with partial polygons placed at the leaves of the tree and the internal nodes are formed by the union of the partial polygons corresponding to its two children. Thus, O(log$m$) phases are required with $m/2$ concurrent polygon union in the first phase and half the number of union in subsequent phases with respect to the previous phase thereby following a reduction tree approach. After each partial polygon union operation, vertices are re-ordered based on the parity of the scanbeam id. If the partial polygons share a common vertex, the merging can be performed in the same way as self-intersections are handled as shown in Figure 3.5. The *virtual* vertices are removed finally by array packing.

3.3.5 Complexity Analysis

The analysis is based on CREW PRAM model. The input for Step 1 is the start and end vertices of both input polygons. Assuming polygon B and O has n vertices each, the sorting of $2n$ vertices can be performed using Cole's $O(\log n)$ time parallel merge sort algorithm [62] using $O(n)$ processors. The input to Step 2 is the set of edges E from both input polygons and the output is m subsets of E where E_i represents edges in the ith scanbeam. Here, m is equal to the number of intervals created by sorting the start and end vertices of edges of both input polygons by y coordinate. Since there are $2n$ vertices, $m < 2n$. Partitioning E into m subsets requires intersection of m scanlines with polygon B and O thereby producing k' *virtual* vertices.

Using segment tree for Step 2: This step can be carried out in two phases using segment tree. The number of edges in the scanbeams need to be counted first. In the second phase, processors need to be allocated based on the count and these processors can report the edges in the scanbeams. To find the edges in a given scanbeam efficiently, a segment tree T is

constructed in parallel with a simple modification that each node stores *count* which is the size of the cover list c. With this modification, the number of edges in a scanbeam can be counted by simply traversing T in $O(\log m)$ time without going through the edges in c. T is queried concurrently by m processors, each processor querying for $y_i \in \{(y_0 + y_1)/2, (y_1 + y_2)/2, .., (y_{m-2} + y_{m-1})/2\}$ in $O(\log m)$ time while keeping track of the indices of the nodes visited and adding the *count* while visiting the nodes. It should be noted that the edges in c are not reported at this stage, instead its count $|c|$ is read. Each query q_i takes y_i as input and returns k_i where k_i is a set of tuples $(k_{i1}, |c_{i1}|)$, $(k_{i2}, |c_{i2}|)$, .., $(k_{ilogm}, |c_{ilogm}|)$, $0 \leq k_{ij} \leq 2^{logm+1}$, where i denotes scanbeam, j denotes the level of the node and k_{ij} denotes the index of the node in T. The number of edges spanning each scanbeam denoted by k'_i is $(|c_{i1}| + |c_{i2}| + .. + |c_{ilogm}|)$. Now we can allocate k' processors in total for Step 2 where k' is $(k'_1 + k'_2 + .. + k'_m)$ for reporting edges in c. The number of processors that can be assigned to a node at k_{ij} is given by $(|c_{0j}| + |c_{1j}| + .. + |c_{logmj}|)$. Since k_i contains the count of edges and their location in T, k' processors can be distributed among the different nodes in T and assigned to the edges in c_{ij} for reporting edges in the scanbeams in $O(\log n)$ time (m is $O(n)$ in the worst case). Since processor requirement is output-sensitive in nature, an $O(\log n)$ time overhead is incurred due to processor allocation.

The input to Step 3 is $O(n+k'+k)$ vertices which are divided among different scanbeams and the number of vertices to be processed in a scanbeam may vary from one scanbeam to another. Let I denote the output size where $I \leq n + k' + k$. Step 3 is executed in parallel for all the scanbeams. Based on *Lemma* 2, vertices lying on top and bottom scanlines are checked if they are contributing vertices in Step 3.1 and 3.3. In *Lemma 3*, it is shown that a prefix sum operation is required to determine contributing vertices for edges of a polygon in a scanbeam in $O(\log n)$. In the worst case, there may be $O(n)$ edges in a scanbeam. As such, four prefix sum operations (two for lower scanline and two for upper scanline) are required for each scanbeam. Since we have allocated $O(n + k')$ processors apriori, this step takes $O(\log n)$ time. From *Lemma 4*, we can allocate k processors in $O(\log n)$ time to report $O(k)$ intersecting vertices and label them in constant time in Step 3.2. Step 3.4 involves

local merging of $O(I)$ vertices by sorting and array packing and can be done in $O(\log I)$ time using $O(I)$ processors. As such, Step 3 can be done in $O(\log I)$ time using $O(I)$ processors. Step 4 can also be done in logarithmic time since it involves sorting of the m partial polygon list by their scanbeam id's and list concatenation only.

Here, we discussed the *intersection* operation but the same analysis holds for *union* and *difference* as well because every intersection of two edges is a vertex of the final output polygon irrespective of the clipping operation performed. Moreover, contributing vertices can be determined using *Lemma* 3 for all clipping operations. Thus, the time complexity of Algorithm 1, dominated by Step 3, is $O(((n+k+k')\log(n+k+k'))/p)$ using $p \leq O(n+k'+k)$ processors. Since k and k' can be no more than $O(n^2)$, we get $O(\log n)$ time algorithm using $(n + k + k')$ processors.

3.4 Multi-threaded Multiway Divide and Conquer Algorithm

In section 3.3, we described PRAM algorithm. This section describes the multi-threaded implementation of Vatti's algorithm. Algorithm 2 describes the algorithm for two input polygons, but also extends to a larger set of polygons. In short, the inputs polygons are partitioned into multiple slabs, each slab is sequentially processed and finally, the partial output is merged together. The start and end vertices of the polygonal edges are sorted first. Then, Minimum Bounding Rectangle (MBR) [1] of the union of input polygons is computed in Step 3. After the partitioning in Steps 4 and 5, partial polygons are sequentially clipped in p horizontal slabs in Step 6. It should be noted that any sequential clipping algorithm can be used in these steps. We employ General Polygon Clipper (GPC) [64] for sequential polygon clipping in Step 6. GPC library is a robust polygon clipping library which takes as input two polygons or two sets of polygons and uses a variation of plane-sweep algorithm as described in [65]. However, in steps 4 and 5, we used Greiner-Hormann (GH) algorithm [59] since we found it to be faster than GPC for rectangular clipping. Step 8 is currently sequentially done as it is a smaller fraction of the time, but can be parallelized as illustrated in Fig 3.6

[1] MBR is represented using bottom-left and top-right vertices with X and Y coordinates.

for stronger scaling.

In order to handle two sets of input polygons, at first, we form the event point list L for plane-sweep by adding the two Y-coordinates of polygonal MBRs in L and sorting it in ascending order. We partition event points in L into p horizontal slabs where p is the number of threads such that every thread gets roughly equal number of local event points. This approach is different from statically segmenting the input polygons in a uniform grid, instead, it relies on the distribution of polygons by taking into account its MBR. Each thread determines the input polygons in its slab. Instead of splitting the polygons spanning consecutive slabs, we replicate those polygons in the slabs. The local event list is readjusted such that no polygon is partially contained in a given slab. Now, the merging phase is not required. However, there may be redundant output polygons which can be eliminated as a post-processing step. Finally, p number of plane-sweep based polygon clipping can be concurrently launched, where p is the number of threads.

Algorithm 2 Multi-threaded Polygon Clipping

Input: polygon A and B with vertices V_A and V_B
clipping operator $op \in \{union,\ intersection,\ difference\}$
$p \leftarrow thread\ count$, $id \leftarrow threadId$
Output: polygon set C
1: add distinct y-coordinates of V_A and V_B to V_y
2: sort V_y
3: rectangle $U \leftarrow compute$ minimum bounding rectangle of $A \cup B$
// partition U into $|V_y|/p$ horizontal slabs
for all slab in parallel **do**
// rect denotes rectangular slab for a thread
4: $A_{id} \leftarrow rectangleClip(A, rect)$
5: $B_{id} \leftarrow rectangleClip(B, rect)$
6: $C_{id} \leftarrow polygonClip(A_{id}, B_{id}, op)$
end for
7: sort the C_{id} polygons by id
8: $C \leftarrow (C_0 \cup C_1 \cup ... \cup C_{p-1})$ // sequential

3.5 Experimental Setup and Implementation Results

To show the performance of multi-threaded algorithms, we present results that we obtained, on both simulated and real data. It contains two versions: one programmed using Java threads for two polygons and another using Pthreads for two sets of polygons. The experiments were performed on 64-bit CentOS operating system running on AMD Opteron (TM) Processor 6272 (1.4 GHz) with 64 CPU cores and 64 GB memory. The Pthread program was compiled using GCC 4.4.6 with O3 optimization enabled.

3.5.1 Synthetic Data

To measure the scalability of Algorithm 2 for a pair of polygons, we created a small test program to produce two polygons namely subject polygon and clip polygon with different number of edges. We generated input polygons with 10, 20, 40, 60 and 80 thousands edges to resemble real-world polygons. The number of intersections between subject and clip polygon was kept linear in the number of edges which represents average case scenario. We ran intersect operation on the simulated pair of polygons to measure the processing time. To obtain some baseline performance, we compared our implementation against Java Topology Suite (JTS) [66], GH algorithm and GPC library. Figure 3.7 shows the performance of sequential polygon clipping algorithms for simulated data. As we can see in the figure, GPC is faster than other clipping algorithms for different sizes of input polygons. However, it takes 32 seconds to intersect input polygons of size 80 thousands each, whereas it takes about 8 seconds to clip polygons of size 40 thousands each. The output size is linear as shown in Figure 3.7. As such, GPC library does not scale well for larger polygons.

Figure 3.8 shows the performance with different data sizes for our multi-threaded implementation. For smaller input sizes, there is no performance improvement. As the size of the input and output grows, the performance also gets enhanced but it saturates at 6 threads due to the parallel overhead. As shown in the figure, we get more than 2 fold speedup when doubling the number of threads for larger datasets. This can be explained from the fact that

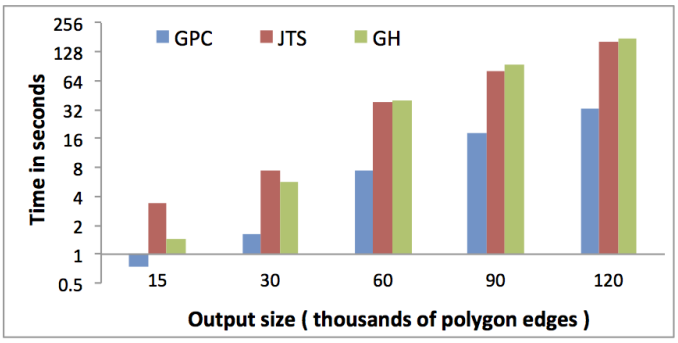

Figure (3.7) *Performance of sequential polygon clipping with varying size of simulated datasets.*

plane-sweep based polygon clipping is a $(n+k)\log n$ time algorithm where k is the number of intersections [3] . Its time complexity can be $O(n\log n)$ in the best case and $O(n^2\log n)$ in the worst case. Both polygons A and B, with n edges each, get split into two partial polygons in Step 4 and 5, with roughly $n/2$ edges each. Hence, the original problem gets decomposed into two subproblems of $(n/2)\log(n/2)$ or $(n/2)^2\log(n/2)$ time complexity for the best case and the worst case respectively. As such, a 4 fold speedup is theoretically possible if partitioning overhead is ignored. The simulated data with $O(n)$ output size can be considered as an average case. Also, the threads use GPC library to solve clipping subproblems and since GPC library is relatively better at clipping smaller polygons in comparison to larger polygons as shown in Figure 3.7, we get better performance due to multi-threading and partitioning of polygons. Practically, we found more than two fold speedup for larger polygons when number of threads is doubled from 1 to 2 and from 2 to 4.

In Figure 3.9, two sets of data (denoted by I and II in the figure) are used to find the execution time for partitioning (Step 4 and 5), clipping (Step 6) and merging (Step 8) as discussed in Algorithm 2. As expected, being the most complex operation, the clipping phase takes more time than partitioning and merging phases. Moreover, the average time taken for partitioning slightly increases as the number of threads increase.

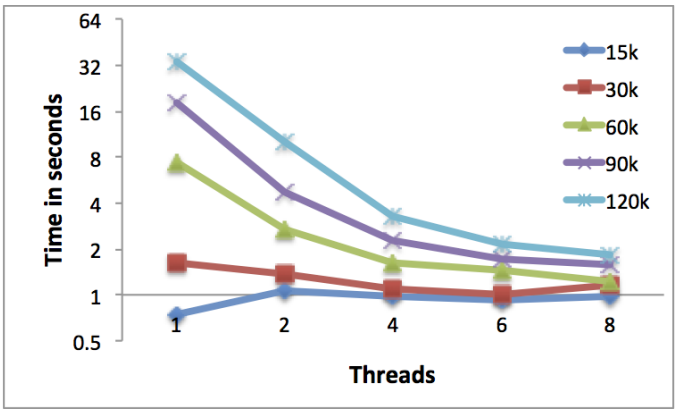

Figure (3.8) *Performance of multi-threaded algorithm for simulated data with different output sizes shown in thousands (k) of edges here.*

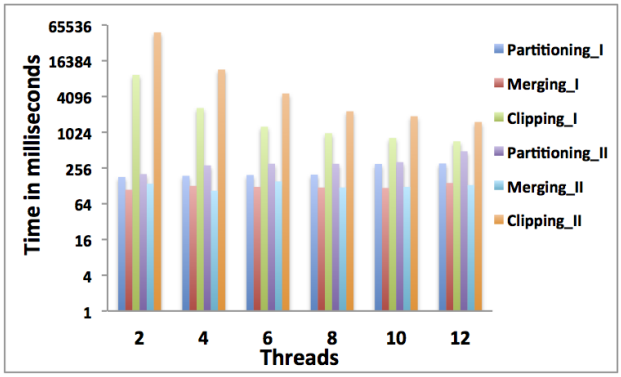

Figure (3.9) *Execution time breakdown of multi-threaded algorithm with simulated datasets. Set I has 80,000 edges and set II has 160,000 edges.*

Table (3.3) *Description of real-world datasets.*

Number	**Dataset**	**Polygons**	**Edges**
1	ne_10m_urban_areas	11,878	1,153,348
2	ne_10m_states_provinces	4,647	1,332,830
3	GML_data_1	101,860	4,488,080
4	GML_data_2	128,682	6,262,858

3.5.2 Real Data

As real-world data, we selected polygonal data from Geographic Information System (GIS) domain. We experimented with publicly available shapefile data. The details of the datasets are provided in Table 6.2. The average edge length for the first dataset is 0.00415 with standard deviation of 0.0101. The average edge length for the second dataset is 0.0282 with standard deviation of 0.0546. We also experimented with GML (Geographic Markup Language) data related to telecommunication domain [67].

As we can see in Figure 3.10, intersection and union of larger datasets (3 and 4 in Table 6.2) scales better than smaller datasets (1 and 2). To obtain absolute speedup, we performed clipping operation using ArcGIS which is state of the art software for GIS datasets. For the same datasets, we also used GPC library but found ArcGIS to be faster. Figure 3.12 shows the absolute speedup gained by our multi-threaded implementation. It took 110 seconds for Intersect (3,4), 135 seconds for Union (3,4) and 28 seconds for Intersect (1,2) operations by ArcGIS. With this baseline performance, we got about 30 fold speedup for Intersect (3,4) and 27 fold speedup for Union (3,4). However, Intersect (1,2) operation when executed sequentially using GPC library, turned out to be about 5 times slower when compared to ArcGIS. It should be noted that we are using GPC library without any preprocessing step. A filter and refine strategy may improve the performance of our multi-threaded algorithm. Intersect (1,2) operation scales well upto 16 threads but only 3.4 fold speedup is obtained. The limited scalability can also be explained by relative load imbalance among threads as shown in Figure 3.11.

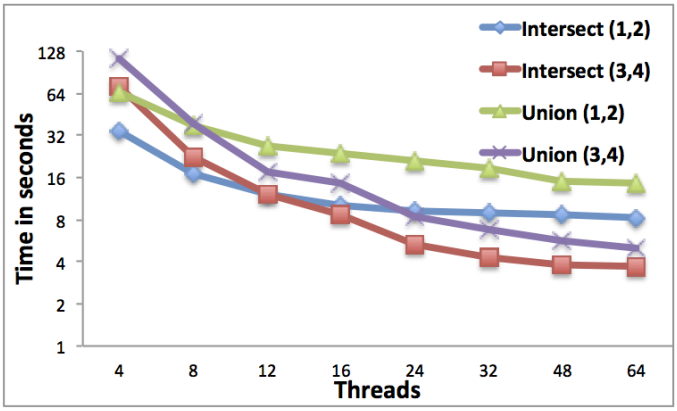

Figure (3.10) *Performance impact of varying number of threads for real-world datasets for Intersection and Union operation.*

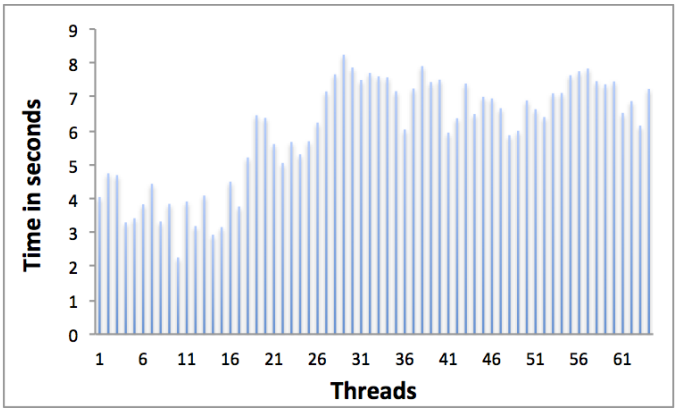

Figure (3.11) *Load imbalance among 64 threads for Intersect (1,2) operation.*

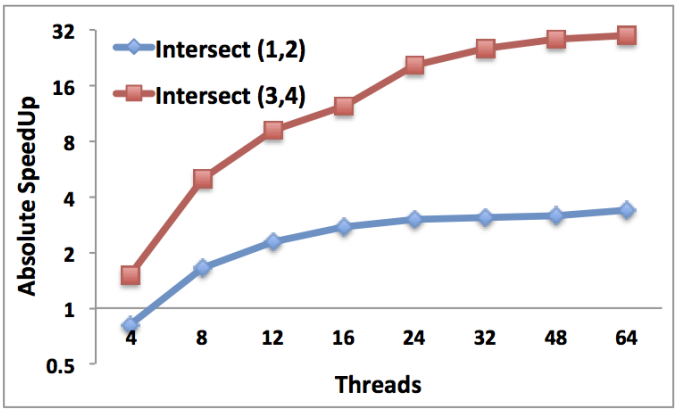

Figure (3.12) *Absolute Speedup for real-world datasets (comparision with ArcGIS software).*

PART 4

PARALLEL GREINER-HORMANN BASED POLYGON CLIPPING ALGORITHM WITH IMPROVED BOUNDS

4.1 Introduction

We begin with some definitions and then introduce the two contributions: parallel Greiner-Hormann algorithm and Parallel MPI-GIS system. In geometry, a polygon is bounded by a finite chain of straight line segments closing in a loop to form a closed chain. A polygon can be classified as a *simple* polygon in case its line segments do not intersect among themselves, otherwise it is termed as *self-intersecting*. A polygon is *convex* if every interior angle is less than or equal to 180 degrees otherwise it is *concave*. An arbitrary polygon may be self-intersecting and concave.

In this paper, we discuss parallel algorithms for two geometric problems involving polygons. First problem is polygon clipping which involves a pair of polygons as input. The second problem originating from Geographic Information System (GIS), is polygon overlay which involves two sets of polygons (also known as polygon layer) as input. In both problems, a geometric set operation, e.g., *intersection and union*, is also specified as input, which dictates how two polygons/layers are combined to generate output polygon(s)/layer. A polygon clipping algorithm can be used to process a pair of overlapping polygons in a polygon overlay operation.

Parallel Greiner-Hormann Algorithm: Parallel polygon clipping algorithms are interesting from time complexity point of view also. We aim to improve the time complexity of existing output-sensitive parallel algorithm. An algorithm is output-sensitive if its running time depends on the size of the output, in addition to the size of the input. In our problem, output-size is the number of vertices in the output polygon which depends on the number of edge intersections found by an algorithm. Sequential output-sensitive polygon clipping

algorithms and sweepline based line segment intersection algorithms have $O((n+k)\log n)$ time complexity. Here, n is the number of vertices in the polygons and k is the number of intersections. In the worst case, k can be $O(n^2)$, but is usually much smaller. In the existing literature, there is no parallel algorithm for polygon clipping with $O((n+k)\log n)$ cost (product of number of processors and parallel time).

Karinthi et al. designed a parallel algorithm for simple polygons which is not sensitive to output size and has $O(\log n)$ time complexity using $O(n^2)$ processors on Exclusive Read Exclusive Write (EREW) PRAM model [68]. We have earlier presented first output-sensitive parallel clipping algorithm based on Vatti's algorithm. Its time complexity using PRAM model is $O((n+k+k')\log n/p)$ where p is the number of processors and k' is the number of additional vertices introduced due to spatial partitioning of polygons [69]. However, its limitation is that it has to process k' additional vertices which in the worst case can be $O(n^2)$. In order to improve the time complexity and reduce the spatial partitioning/merging overhead, we have explored sequential Greiner-Hormann (GH) algorithm for parallelization in this paper. Our parallel Griener-Hormann algorithm can perform clipping in $O(\log n)$ time using $O(n^2)$ processors for arbitrary polygons with self-intersections. It should be noted that in our algorithm, self-intersecting polygons need not be broken into simple polygons. This is important because there can be $O(n^2)$ self-intersections. An output-sensitve version of our Greiner-Hormann algorithm can perform clipping in $O(\log n)$ time using $O(n+k)$ processors.

GH algorithm is suitable for Graphics Processing Unit (GPU) parallelization. We have provided multi-core and many-core parallelization of GH algorithm. We also provide complexity analysis of our parallel algorithm and prove that its cost matches the time complexity of sequential sweepline based polygon clipping.

MPI-GIS: Distributed Overlay Processing System: Real world polygons can be large in size; for example, one polygon representing ocean in shapefile (*ne_10m_ocean.shp*) has more than 100,000 vertices. When large volumes of data are deployed for spatial analysis and overlay computation, it is a time consuming task, which in many cases is also time sensitive. For emergency response in the US, for example, datasets used by HAZUS-MH [70] have

the potential to become very large, and often beyond the capacity of standard desktops for comprehensive analysis, and it may take several hours to obtain the analytical results [71]. Although processing speed is not critical in typical non-emergency geospatial analysis, spatial data processing routines are computationally intensive and run for extended periods of time.

In VLSI CAD, polygon intersection and union operations are used for design rule checking, layout verification, etc. In a paper published by Intel in 2010 regarding large scale polygon clipping for microprocessor chip layout, clipping libraries reportedly took 4.5 hours to several days [3]. As such, the challenges are excessive run-time, excessive memory consumption, and large data size.

We had previously developed a preliminary GIS overlay system using Message Passing Interface (MPI) to tackle some of these challenges [17,71]. It had a master-slave architecture where master and slave processes read the entire file sequentially and master process was responsible for task creation using sequential R-tree construction. As such, its scalability was limited. In this paper, learning from the bottlenecks, we present a newly designed *MPI-GIS* system which has led to a significantly improved scalability.

Contributions: Main contributions in this paper can be summarized as follows:

- An output-sensitive PRAM algorithm for clipping a pair of *simple* polygons in $O(\log n)$ time using $O(n+k)$ processors, where n is the number of vertices and k is the number of edge intersections. For self-intersecting polygons, the time complexity is $O(((n+k)\log n \log\log n)/p)$ using $p \leq (n+k)$ processors. Using $\Theta(n^2)$ processors, our algorithm also achieves $O(\log n)$ time complexity but unlike Karinthi's [68], it can also handle self-intersections.

- Multi-threaded implementation of parallel polygon clipping based on Greiner-Hormann algorithm [59],

- GPU implementation of Greiner-Hormann polygon clipping using CUDA,

- A practical polygon overlay system (*MPI-GIS*) for real-world GIS data yielding about 44X speedup on a IBM iDataPlex cluster [72] with 32 compute nodes, each node

having 8 cores, while processing about 600K polygons as compared to ArcGIS version 10.1 running on Windows 7 machine having 2.4 GHz Intel Core i5 processor and 4 GB memory.

The paper's organization is as follows: Section 4.2 briefly reviews PRAM model, polygon clipping algorithms, and its time complexity. Section 4.3 describes the parallel polygon clipping algorithm based on Greiner-Hormann's vertex list traversal algorithm [59]. Section 5.4 presents our *MPI-GIS* distributed overlay system. In Section 4.4, experiments conducted to evaluate the multi-threaded implementation of parallel GH algorithm and *MPI-GIS* system performance are described.

4.2 Background Concepts and Related Work

4.2.1 NVIDIA Graphics Processing Unit

Modern General-purpose graphics processing unit (GPGPUs or GPUs) are fully programmable many-core graphic processing units. Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed for parallel processing. A typical CUDA program is organized into host programs and one or more parallel kernel programs. The host programs are executed on CPU while the parallel kernel programs run on GPUs. Java programs can use *jcuda* which is a Java binding for CUDA in order to communicate with GPU. In this case, the host program is written using Java.

A GPU consists of an array of parallel processors which are often referred to as streaming multi-processors (SM). Each SM also has small on-chip shared memory in KBs which has lower access latency and higher bandwidth compared to device memory which is accessible to every SM, and has larger size. The SMs employ a Single Instruction Multiple Thread (SIMT) architecture. A group of 32 threads called a warp is the minimum execution unit. Once scheduled on a SM, the threads in a warp share the same instruction and can execute in a fairly synchronous fashion.

4.2.2 Sequential Polygon Clipping of Arbitrary Polygons

There are only a few algorithms in literature which can handle arbitrary polygons. Vatti's algorithm [2] and GH algorithm are two important algorithms among them [63]. In GH algorithm, polygons are represented by doubly-linked lists of vertices. In order to clip the subject polygon S against the clip polygon C with m and n vertices respectively, first step is to find intersections using a naive $O(mn)$ algorithm and label each as *entry/exit*. Using traversal of vertices guided by the labels, first, the part of the boundary of S in C is found out, then the part of the boundary of C in S, and then these two parts are combined. On the other hand, Vatti's algorithm uses a variation of sweepline technique to find intersections quickly which is an improvement over GH algorithm. The output polygon is also constructed as the intersections are found by scanning the vertices from bottom to top.

4.2.3 Differences between GH and Vatti's algorithm

Apart from the difference in algorithms for finding edge intersections, one important difference between the two algorithms is that GH algorithm is not output-sensitive and it has a time complexity of $O(mn)$ for clipping a pair of polygons with m and n vertices. On the other hand, Vatti's algorithm is sensitive to output size. The second difference is the way output polygons are formed. For self-intersecting polygons, Vatti's algorithm requires $(\binom{m}{2} + \binom{n}{2} + mn)$ intersections in the worst case to construct output polygon(s), since it includes self-intersections to form output polygon. On the other hand, GH algorithm requires only mn intersections to construct the output [59]. As an example, for the polygons in Figure 4.1, one of the output polygons generated by Vatti's algorithm includes 13 vertices $\{S4, S3, I8, S6, I6, I5, I4, I3, C4, I5, I7, I8, S5\}$. Whereas, GH algorithm generates 9 vertices only $\{S4, S3, I7, I4, I3, C4, I6, S6, S5\}$ to represent the output polygon. This is an advantage of GH algorithm over Vatti's algorithm.

4.2.4 Time Complexity

It has been shown that polygon intersection for simple polygons can be carried out in $\Theta(n\log n + mn + m\log m)$ time sequentially [73] and this cannot be improved further (in algebraic tree model of computation). Sweepline based polygon clipping has time complexity of $O((n+k)\log n)$ where k is the number of intersections [3]. Karinthi et al. have designed a parallel algorithm for simple polygons which is also not sensitive to output size and has $O(\log n)$ time complexity using $O(n^2)$ processors on Exclusive Read Exclusive Write (EREW) PRAM model [68].

4.3 Parallel Greiner-Hormann Polygon Clipping Algorithm and Implementations

4.3.1 Terminology

The input consists of two self-intersecting polygons namely subject polygon (S) and clip polygon (C) as shown in Figure 4.1. The polygons are represented as a doubly linked lists (implemented using arrays) with each vertex having *prev* and *next* links. Along with polygons, the input also specifies a geometric operator namely *union or intersection*. *Union* operation generates new polygon(s) containing the area inside either of the two polygons. *Intersection* operation generates new polygon(s) containing the area inside both polygons. When two polygons are overlaid to produce the output polygon, new intersection vertices are produced. Some edges/vertices will be part of the output and we call them as *contributing* edges/vertices.

Intersection between edges belonging to S and C can be labeled as *entry* and *exit* with respect to the interior of the polygon. An intersection is labeled as *entry* if the edge of S (or C) intersecting C (or S) traverses from exterior to the interior of C (or S). Similarly, an intersection is labeled as *exit* if the edge of S (or C) intersecting C (or S) traverses from interior to the exterior of C (or S). In the Figure 4.1, $I1$ is an *entry* vertex and $I2$ is an *exit* vertex. This is computed using point-in-polygon test [74] which checks if a vertex of S (or C)

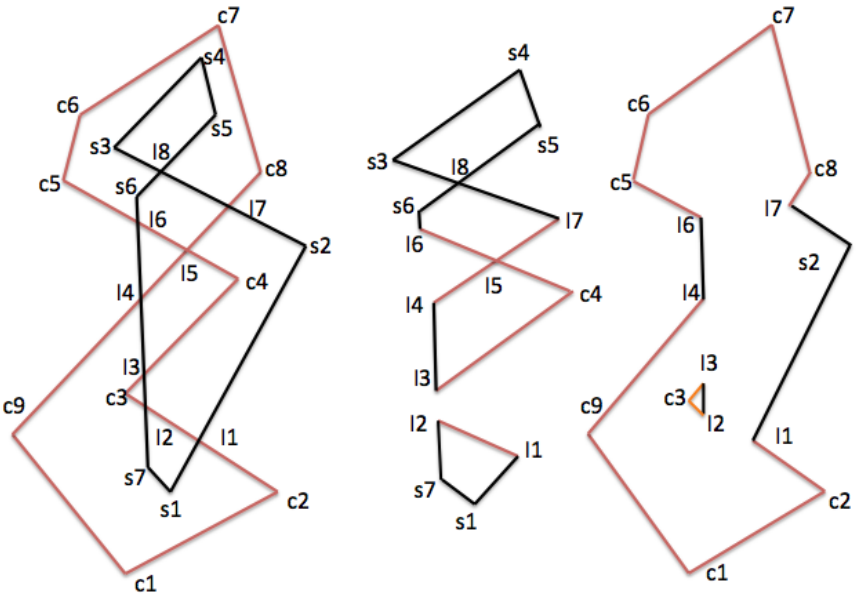

Figure (4.1) *Example showing intersection and union of two self-intersecting concave polygons.*

lies within C (or S) or not. In addition to a label, an intersection vertex has two parameters α_s and α_c corresponding to S and C respectively, which indicates where the intersection point lies relative to the start and end point of intersecting edges [59]. The value of α_s and α_c is in between 0 and 1. For edge s_1s_2 intersecting with edge c_1c_2, α_s can be calculated as follows:

$$\alpha_s = \frac{\delta_{cx}(s_1.y - c_1.y) - \delta_{cy}(s_1.x - c_1.x)}{\delta_{cy}(s_2.x - s_1.x) - \delta_{cx}(s_2.y - s_1.y)}$$

Here, $\delta_{cx} = (c_2.x - c_1.x)$ and $\delta_{cy} = (c_2.y - c_1.y)$.

As we can see in Figures 4.1 and 4.4, edge C_2C_3 has two intersection I_1 and I_2 which needs to be inserted in between C_2 and C_3 in sorted order based on a parameter α derived from the coordinates of intersecting edges C_2C_3, S_1S_2, and S_6S_7.

GH algorithm uses the labels and α value of intersections to sequentially traverse the vertices of input polygons in order to select contributing vertices only. This labeling scheme can be adapted to a parallel setting also where S and C may be partitioned into multi-

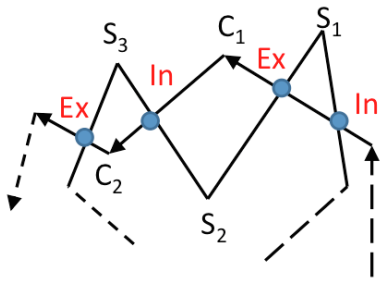

Figure (4.2) *Two partial polygonal chains* (S_1, S_2, S_3) *and* (C_1, C_2) *are intersecting here. Four edge intersections are labeled with* entry(In) *or* exit(Ex).

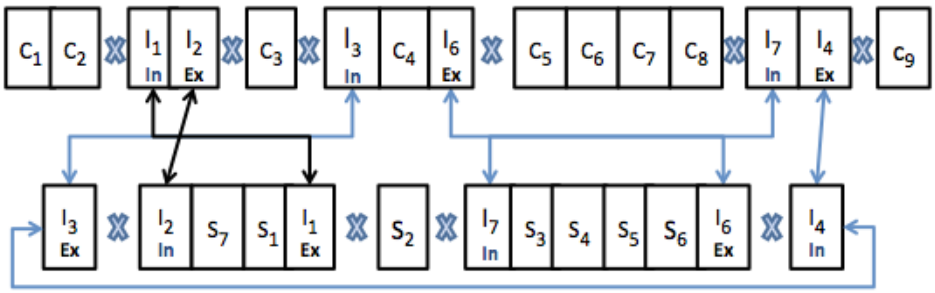

Figure (4.3) *Intersection by manipulating links of intersections based on* entry*(In)/*exit*(Ex) labels. A processor is assigned to each intersection which locally determines the connectivity to the neighboring vertices. A cross shows nullification of a link.* Intersection *produces two output polygons namely* $P_1 = \{(I_1, I_2, S_7, S_1, I_1)\}$ *and* $P_2 = \{I_3, C_4, I_6, S_6, S_5, S_4, S_3, I_7, I_4, I_3\}$. *An output polygon forms a closed loop. The black and blue colors are used to differentiate between two output polygons.*

ple chains of vertices as shown in Figure 4.2. *Lemma 1* shows how labeling can be done independently (Figure 4.2).

Lemma 1: Labeling Intersections: Given a partial polygon chain C' from C intersecting with S, intersections in the chain can be labeled independently as *entry* or *exit* vertices.

Proof sketch: Intersection vertices of a given edge in C' can be ordered using its α value independently of the other edges. Greiner-Hormann *entry/exit* labeling scheme holds for ordered list of intersections of a single edge as well as for a collection of edges in C'. In a given chain of intersections, the label of only the first intersection needs to be determined using point-in-polygon test. For the remaining ones, labeling can be done locally without going through all the edges of S. □

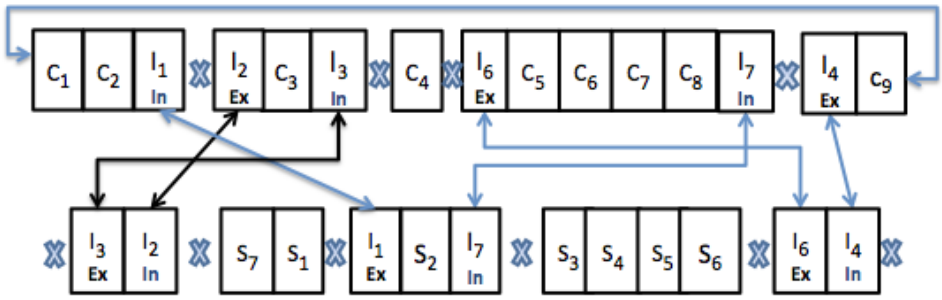

Figure (4.4) *Union by manipulating links of intersections based on* entry*(In)/*exit*(Ex) labels.* Union *produces P3* $=\{C_1, C_2, I_1, S_2, I_7, C_8, C_7, C_6, C_5, I_6, I_4, C_9, C_1\}$ *and P4*$=\{I_2, C_3, I_3\}$. *An output polygon forms a closed loop. The black and blue colors are used to differentiate between two output polygons.*

Intersection vertices are always part of the output irrespective of the geometric operator. A vertex from the input polygons can be checked if it is *contributing* or not based on the label of the two intersections bounding them. If we consider *intersection* operation, *contributing* vertices are bounded by intersections with In and Ex labels. Using this insight, an intersection vertex can independently manipulate its neighboring links as shown in *Lemma 2* (Figure 4.4).

Lemma 2: Link nullification rule: For *intersection* operation, if an intersection vertex has *entry* label, then its *prev* link is nullified. If an intersection vertex has *exit* label, then its *next* link is nullified. For *union* operation, the rule is opposite to that of *intersection*.

Proof sketch: This simple rule works since by the definition of the labels, it can be shown that vertices of S between intersections with *entry* and *exit* label are interior to C and thus *contributing*. Similarly, it can be shown that vertices of S between two consecutive intersections with *exit* and *entry* label are exterior with respect to C and thus non-contributing. On the contrary, the exterior region is required for *union*. □

In order to label an intersection vertex, point-in-polygon test can be done for a vertex v of the intersecting edge of a polygon with respect to the other polygon P. This test has $O(n)$ sequential time complexity [74]. A simple parallel point-in-polygon testing is possible.

Lemma 3: PRAM Point-in-Polygon test: Given a vertex v of an edge intersecting with P, it can be tested if it is inside P in $O(\log n)$ time using n processors.

Proof sketch: Draw a ray Y starting from v and count the number of edges E of P that intersect ray Y. Assign a processor to $e \in E$. For all e_i, $i \in \{1, .., |n|\}$ in parallel, a processor increments its *count* if e_i intersects Y and is not collinear with Y. Using n processors, sum the individual *count* of all the processors using reduction tree approach in $O(\log n)$ time. If the count is odd, then v is inside P. □

PRAM Point-in-Polygon test is carried out for a single intersection vertex of S and C.

Lemma 4: Intersection labeling in $O(\log n)$ time: Given a vertex list of a polygon with k intersection vertices, *entry/exit* labeling of these vertices can be done in $O(\log n)$ time using $O(max(n, k))$ processors on PRAM model.

Proof sketch: From *Lemma 1*, we know that consecutive intersections have alternating labels in a polygon. As such, it suffices to find the label of one intersection only (say i) using *Lemma 3* in $O(\log n)$ time and use it to label the remaining ones. Assuming that the intersections are ordered as they appear in the polygon, if the intersections are enumerated from 1 onwards, odd numbered intersections will have the same label as i and even numbered intersections will get the opposite label as that of i. Each one among k processors can label itself based on the number assigned to it in constant time. As such, $O(max(n, k))$ processors are sufficient. □

Now, we present the Parallel Greiner-Hormann clipping algorithm for two self-intersecting polygons S and C.

4.3.2 PRAM Polygon Clipping Algorithm

For simplicity, we assume that any vertex of one polygon does not lie on an edge of the other polygon. Such degenerate cases can be handled by perturbing vertices slightly as shown in [59]. The gist of the algorithm is to find all intersections between edges of S and C and insert them into proper positions in the list of vertices in parallel.

The algorithm is shown in Algorithm 3. In case S is contained in C or vice versa, Step 1 will return 0 intersections and *Lemma 3* is used to determine if one polygon is contained inside another. After the algorithm terminates, *contributing* vertices get linked together and

Algorithm 3 Parallel Griener-Hormann Polygon Clipping

Input: $S(V_S, E_S)$ and $C(V_C, E_C)$. $n = |E_S| + |E_C|$

1. Find edge intersections I between E_S and E_C. Add I to V_S and to V_C. Generate tuples (e_s, i, α_i^s) and (e_c, i, α_i^c) where i is an intersection between edges e_s and e_c.

2. For each edge in the input polygons, generate tuples $(e_s, l_s, |l_s|)$ and $(e_c, l_c, |l_c|)$ where l_s, l_c denotes list of intersections and $|l_s|$, $|l_c|$ denotes number of intersections for a given edge. Sort l_s and l_c based on α_s and α_c respectively. Insert l_s into e_s. Similarly, insert l_c into e_c

3. Create a link between intersection $i \in V_S$ and $i \in V_C$ (e.g., as shown between I_1 with label In and I_1 with label Ex in Figure 4.4).

4. Assign *entry* or *exit* label to intersections in V_S and V_C.

5. Store the indices and labels of intersections in V_S and V_C in array L_S and L_C respectively.

6. Using the *entry/exit* labeling, each intersection vertex locally nullifies its link to adjacent non-contributing vertex using link nullification rule.

7. Using L_S and L_C arrays, non-contributing vertices are removed.

non-contributing vertices get eliminated.

Complexity Analysis: We use CREW PRAM model for analysis. If a naive algorithm is used in Step 1, using $O(n^2)$ processors, all k intersections can be found out in constant time by assigning a processor to each pair of edges and computing all possible intersections. Using $O(n^2)$ processors is suitable in the worst case scenario with $O(n^2)$ intersections. However, the processor requirement is high considering that there may be constant or $O(n)$ intersections only in some scenarios. When k is small, an output-sensitive parallel algorithm is more applicable. Using $O(((n+k)\log n \log\log n)/p)$ time algorithm [41] with p processors ($p \leq (n+k)$), k intersections can be reported. The value of k is determined on-line by this algorithm. If the polygons are not self-intersecting *(simple)*, then in this case, the time complexity can be improved to $O(\log n)$ using $O(n + k/\log n)$ processors [44]. Since k is determined in the first step, the number of processors required for the remaining steps is $O(n+k)$. Using $O(k)$ processors and the number of intersections $|l_s|$ and $|l_c|$ associated with a given edge, Step 2 can be done in $O(\log k)$ time using Cole's merge sort algorithm [62]. Since, k can be $O(n^2)$ in the worst case, Step 2 takes $O(\log n)$ time. Step 4 takes $O(\log n)$ time using Lemma 3. In Step 5, list ranking is used to index the vertices in V_S and V_C in $O(\log n)$ time [75]. Using vertex index as key, binary search is performed in the array L_S and L_C to identify non-contributing vertices. Step 6 can be done in constant time using k processors. Step 7 takes $O(\log n)$ time using $O(max(n,k))$ processors.

Dominated by Step 1, the time complexity for polygon clipping for simple polygons is $O(\log n)$ using $(n+k)$ processors. For self-intersecting polygons, the time complexity is $O(((n+k)\log n \log\log n)/p)$ using $p \leq (n+k)$ processors. Using a naive algorithm for Step 1, the time complexity of Algorithm 3 is $O(\log n)$ using $O(n^2)$ processors for arbitrary polygons with self-intersections. This time complexity is same as Karinthi's which can only handle simple polygons.

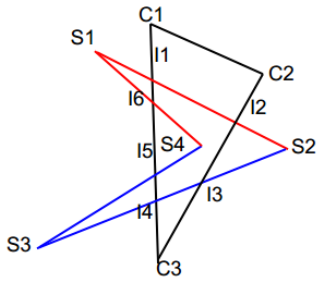

Figure (4.5) *Intersection of polygons $S = \{S1, S2, S3\}$ and $C = \{C1, C2, C3\}$ using 2 threads. The edge intersections are $\{I1, I2, .., I6\}$. Thread 1 gets red-colored edges and finds (I1, I2, I6) intersections. Thread 2 gets blue-colored edges and finds (I3, I4, I5) intersections.*

4.3.3 Multi-threaded implementation design

In the previous subsection, we demonstrated the maximum inherent parallelism in polygon clipping by presenting the algorithm using PRAM model and focused on its time complexity. Here, we discuss the implementation details for multi-threaded parallelization for multi-core CPU and many-core GPU. The experimental results are in Section 4.4. The intersection reporting task is distributed among threads by statically partitioning the edges of S among available threads. C is available to all the threads in global memory. Each thread determines its portion from m edges of polygon S as shown in the example given in Figure 4.5. There are different ways of executing Step 1. Since there can be (mn) intersections in the worst case, the straightforward method would be to allocate p threads in such a way that a given thread is responsible for roughly $((m/p) \times n)$ intersections. This naive way of reporting all intersections may be suitable for polygons with relatively fewer number of edges, but might be an overkill for polygons with larger number of edges even with multi-core parallelization. However, many-core parallelization using GPU might be faster.

Multi-core implementation: Algorithm 3 is first carried out for S by all the threads in parallel. In the example given in Figure 4.5, we can see that all the intersections for a given edge from S are found locally. But, in the case of C, for $(C3, C1)$ edge, thread 1 is able to find $I1$ and $I6$ only. On the other hand, thread 2 is able to find $I4$ and $I5$ only. So, a barrier synchronization is required after intersections are reported by the threads. After

the synchronization phase, all the intersections are available for a given edge in C. Then, intersections are sorted, labeled, and contributing/non-contributing vertices are identified. Finally, the partial output from the two polygons are merged together by traversing the vertices of the input polygons.

GPU Acceleration: Edge intersections reporting computation can be accelerated using Single Instruction Multiple Thread (SIMT) parallelism. At first, the vertices of the input polygons S and C are transferred to GPU device memory. Due to limited support for dynamic memory allocation on GPU, number of intersections are counted first, so that memory can be allocated on GPU to store the intersections. In order to take advantage of fast but smaller shared memory, the vertices of the polygons are partitioned into tiles that can fit into the shared memory, and loaded from device memory to the shared memory. Overall clipping operation is written in terms of three functions (known as kernels). In the first kernel, for each edge in S, a GPU thread checks if the edge intersects with all the edges of C, counts the number of intersections, and stores it in variable *count*. Then, using prefix sum of *count* for all the threads, each thread determines the relative location to store intersections in GPU memory. There are two block-level synchronization used in this kernel, one after the vertices are loaded into shared memory and the second synchronization is required before prefix sum calculation. The second kernel stores the intersections along with α values and indices of the intersecting edges in device memory. In the third kernel, each thread performs point-in-polygon test for a vertex in S and C to find contributing vertices. Finally, the result is transferred back to host memory for further processing.

4.4 Experiment and Evaluation

To show the performance of multi-threaded clipping algorithms, we present results that we obtained, on both simulated and real data in Subsection A. The evaluation of *MPI-GIS* is presented in Subsection B.

4.4.1 Multi-threaded Parallel Greiner-Hormann Clipping Algorithm Evaluation

Hardware Platform: Single-threaded and multi-threaded experiments were run on a 2.33 GHz Intel Xeon processor with 8 cores and 32 GB of memory. We used Java multi-threading for implementation. For GPU acceleration, NVIDIA Tesla C2075 is used. CUDA 5.5 and *Jcuda* is used for C and Java-based implementations, respectively. *Jcuda* is a Java binding for CUDA [76].

Spatial data: We tested our multi-threaded implementation using two input polygons with different number of intersections (denoted by k). Depending on k, there are three test cases namely i) k=$O(1)$, ii) k=$O(n)$, and iii) k=$O(n^2)$. To measure the scalability of multi-threaded implementation, we created a small generator program to produce two polygons S and C with different number of vertices. We generated input polygons with vertices ranging from 1K to 100K (K is thousands). The number of vertices in S and C are equal. The number of intersections is equal to the number of vertices in S. To test the worst case scenario, we generated another set of polygons with n vertices in S and m vertices in C with $k = O(nm)$ intersections.

For real-world data, we used *Classic* dataset [77] which has two polygons with about 100K and 70K vertices. The number of intersections is about 50 only and the number of vertices in the output polygon is about 37K. We also used large polygons from *Ocean* dataset (*ne_10m_ocean shapefile*) which has two polygons representing an ocean with 100K and a continent with 16K vertices. The number of intersections is about 10K and the number of vertices in the output polygon is about 50K in this case.

To obtain some baseline performance, we compared our implementation against Java Topology Suite (JTS) [78], GH algorithm and General Polygon Clipper (GPC) library [79]. GPC is an implementation of Vatti's algorithm. We ran *intersect* operation on the simulated pair of polygons to measure the processing time. Figure 4.6 compares the execution time of sequential polygon clipping libraries using simulated data for polygons with $k = O(n)$ intersections. As we can see in the figure, in general, GPC is faster than other clipping algorithms for different sizes of input polygons. However, in the worst case scenario, when k is

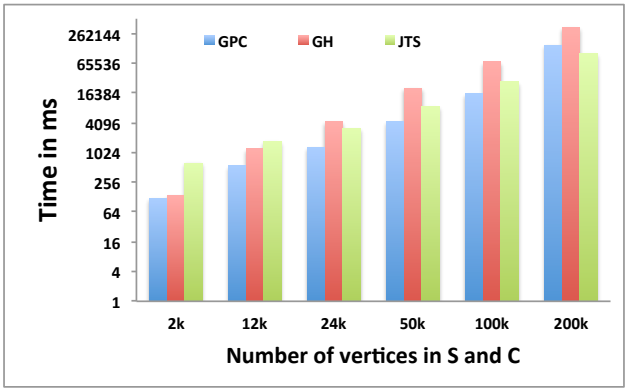

Figure (4.6) *Execution timing for 3 sequential polygon clipping implementations using simulated datasets having $O(n)$ intersections.*

$O(n^2)$ $(m = n)$, GH algorithm is the fastest as we can see in Figure 4.7. This can be explained by comparing the computational complexity of GPC which uses a variation of sweepline technique and GH algorithm which uses $O(n^2)$ algorithm for reporting intersections. The time complexity of sweepline algorithm is $O((n + k)\log n)$. So, when k is $O(n^2)$, GPC takes $O(n^2 \log n)$ time.

Table (4.1) *Execution timings in ms for Parallel GH using simulated datasets with varying number of vertices (n) in S. C has 10 vertices (m). There are $O(nm)$ intersections.*

Threads	**1k**	**6k**	**12k**	**25k**	**50k**
1	139	379	480	612	858
4	128	354	443	545	715
8	123	341	420	505	696

We developed two versions, namely *Parallel GH* which is a multi-threaded implementation and *CUDA-GH* which uses GPU for fast edge intersection reporting and to find contributing vertices in S and C. Figure 4.8 shows the execution time for *Parallel GH* using polygons with $k = O(n)$ intersections. For smaller input sizes, there is no perfor-

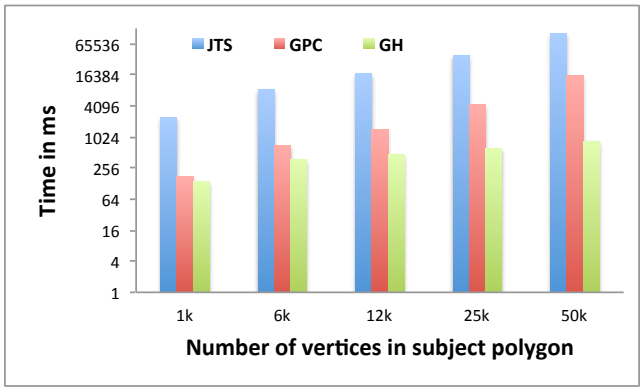

Figure (4.7) *Execution timing for 3 sequential polygon clipping implementations using simulated datasets with varying number of vertices (n) in S. C has 10 vertices (m). There are $O(nm)$ intersections.*

mance improvement. For polygons with more than 12K vertices, *Parallel GH* scales well upto 8 threads. Table 4.1 shows the execution time for *Parallel GH* using polygons with $k = O(nm)$ intersections. In this case, every edge of S intersects with all the edges of C. Although intersection reporting task is distributed evenly among threads, the scalability is still limited. This is due to the sequential creation of quadratic number of new edges of the output polygon which dominates overall execution time of *Parallel GH*.

Figure 4.9 shows the time taken by the three kernels and the setup/initialization overhead in *CUDA-GH* using polygon sets with different number of vertices. The setup overhead is caused by using *Jcuda*, establishing context with GPU, memory allocation, and memory copy operations. The setup overhead is very high when compared to the time taken for counting and reporting intersections. The time taken by individual kernels increases with the increase in the number of vertices in the polygons. But, the setup time is almost the same irrespective of the size of the datasets.

Figure 4.10 shows the comparison between *Parallel GH* and *CUDA-GH* for simulated datasets with $O(n)$ intersections. For the polygon with 24K vertices or less, *Parallel GH* performs better than *CUDA-GH*. This is due to the high setup overhead incurred by *CUDA-GH* while using GPU. Effectively, the runtime is I/O bound between CPU memory and GPU

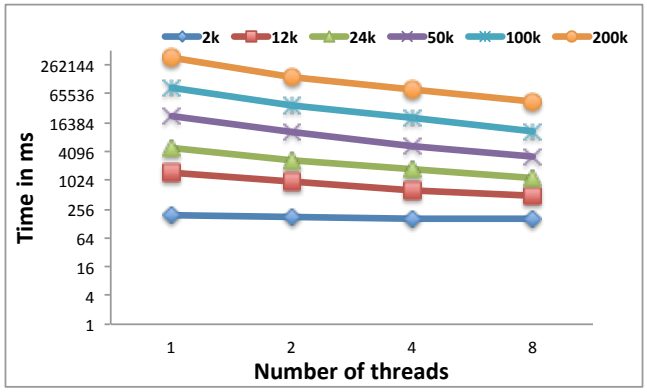

Figure (4.8) *Execution timing for Parallel GH using simulated datasets of different sizes of subject and clip polygons with $O(n)$ intersections.*

memory for smaller datasets. However, as the number of vertices in the input polygons become larger than 50K, *CUDA-GH* benefits from fast intersection reporting using GPU acceleration and thus performs better.

Table (4.2) *Execution timing in ms for GPC, CUDA-GH and Parallel GH using 8 CPU cores. For Classic+Ocean, to measure GPC time, two threads are used. First thread clips Classic and second thread clips Ocean.*

Polygons	**GPC**	**CUDA-GH**	**Parallel GH**
Classic	9, 047	6, 192	80, 031
Ocean	3, 547	3, 311	15, 772
Classic+Ocean	9, 101	7, 041	95, 803
Simulated (50k)	4, 273	2, 803	3, 203
Simulated (100k)	16, 005	3, 613	10, 790
Simulated (200k)	151, 365	7, 196	45, 158

In Table 4.2, we compare *Parallel GH*, *CUDA-GH*, and GPC library using real-world and simulated polygons. For comparison, we used GPC because it is faster than sequential

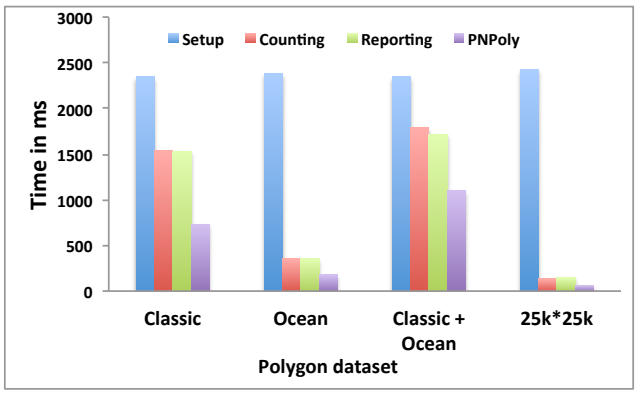

Figure (4.9) *Execution time breakdown for CUDA-GH running on Tesla GPU using real-world and simulated datasets. In the third dataset, Ocean and Classic polygons are combined and processed simultaneously. 25K*25K data has about 25K intersections.*

GH implementation for real-world and simulated datasets with $O(n)$ intersections, as we can see in Figure 4.6. *Parallel GH* is faster than GPC for simulated polygons whereas it is slower than GPC for real-world polygons. This is due to the fact that GPC is faster when the number of intersections are less compared to the number of vertices in the polygon. The real-world polygons have less intersections, 10 in case of *Classic* and 20K in case of *Ocean* which is only one-tenth of the number of vertices in S. With GPU acceleration, there is 4 to 12 times improvement for real-world polygons in comparison to *Parallel GH*.

For *Ocean* dataset, *CUDA-GH* is slightly faster than GPC. As we can see in Figure 4.9, the setup cost dominates the overall clipping time. If we remove the GPU setup overhead, then it only takes less than 1 second for *ocean* data which is much faster than the time taken by GPC. To study the effect of setup overhead in case of more than one pair of polygons, first we measured the execution time for *Classic* and *Ocean* separately in the first two rows of Table 4.2, and then we combined the two in the third row (*Ocean* + *Classic*). For the combined dataset, the two pairs of polygons are serialized, copied to GPU, and clipped together. For the combined dataset, *CUDA-GH* takes about 7 seconds which is faster than clipping the two pairs of polygons separately. So, it is possible to get better performance from GPU by clipping multiple pairs of polygons together. Nevertheless, for clipping a pair

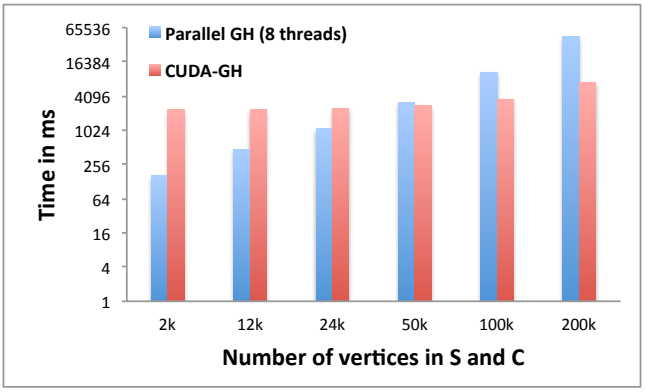

Figure (4.10) *Execution time for Parallel GH, CUDA-GH running on Tesla GPU and GPC using simulated datasets having $O(n)$ intersections.*

of polygons, it is possible that the overhead of using GPU may be greater than the time it takes to simply process the polygons sequentially on the CPU directly.

Comparison with Parallel Vatti Algorithm: Our previous multi-core implementation was based on spatial partitioning of input polygons. In each partition, we used sequential clipping. However, partitioning and merging overhead limited its scalability. Moreover, it was difficult to implement on GPU since it utilizes sequential sweepline technique to find edge intersections.

PART 5

MPI-GIS : GIS POLYGONAL OVERLAY PROCESSING USING MPI

5.1 Introduction

Scalable vector data computation has been a challenge in Geographic Information Science and Systems (GIS). When large volumes of data are deployed for spatial analysis and overlay computation, it is a time consuming task, which in many cases is also time sensitive. For emergency response in the US, for example, disaster-based consequence modeling is predominantly performed using HAZUS-MH, a FEMA-developed application that integrates current scientific and engineering disaster modeling knowledge with inventory data in a GIS framework [80]. Depending on the extent of the hazard coverage, datasets used by HAZUS-MH have the potential to become very large, and often beyond the capacity of standard desktops for comprehensive analysis, and it may take several hours to obtain the analytical results. Although processing speed is not critical in typical non-emergency geospatial analysis, spatial data processing routines are computationally intensive and run for extended periods of time. In addition, the geographic extents and resolution could result in high volumes of input data.

In this paper, we present a parallel system to execute traditional polygon overlay algorithms on a Linux cluster with InfiniBand interconnect using MPI framework.

We present the algorithm for carrying out the overlay computation starting from two input GML (Geography Markup Language) files, their parsing, employing the bounding boxes of potentially overlapping polygons to determine the basic overlay tasks, partitioning the tasks among processes, and melding the resulting polygons to produce the output GML file. We describe the software architecture of our system to execute the algorithm and discuss the design choices and issues. Our instrumenting of the timing characteristics of various phases of the algorithm rigorously point out portions of the algorithm which are

Source	Example Type	Description	File Size
US Census	Block Centroids	Block centroids for entire US	705 MB
	Block Polygons	2000 Block polygons for the state of Georgia	108 MB
	Blockgroup Polygons	2000 Blockgroup polygons for the state of Georgia	14 MB
GADoT	Roads	Road centerlines for 5-county Atlanta metro	130 MB
USGS	National Hydrography Dataset	Hydrography features for entire US	13.1 GB
	National Landcover Dataset	Landcover for entire US	3-28 GB
JPL	Landsat TM	pan-sharpened 15m resolution	4 TB
Private	LIDAR	LIDAR point clouds 1-4 pts/sq. ft	0.1-1 TB

Figure (5.1) *Example GIS Datasets and Typical File Size Ranges.*

easily amenable to scalable speedups and some others which are not. The latter is primarily related to file related I/O activities. The experiments also point out the need for input and output GML files to be stored in a distributed fashion (tranparent to the GIS scientists) as a matter of representation to allow efficient parallel access and processing.

Our specific technical contributions are as follows:

- Porting the Windows Azure cloud-based spatial overlay system to Linux cluster using MPI
- Implementing and improving an end-to-end overlay processing system for a distributed cluster
- An absolute speedup of over 15x using 80 cores for end-to-end overlay computation over moderate sized GML data files of 770 MB intersected with 64 MB file with skewed load profile.

The rest of this paper is organized as follows: Section 6.2 reviews the literature briefly and provides background on GIS raster and vector data, various operations that define parallel overlay, and R-tree and general polygon clipper (GPC) library. Section 5.3 describes two flavors of task partitioning and load distribution. Several key implementation related issues are discussed in Section 5.3.4. Our experimental results and other experiments are in 6.4.

5.2 Background and Literature

5.2.1 Data Types in GIS

In GIS the real world geographic features are prominently represented using one of the two data formats: raster and vector. Raster form stores the data in grids of cells with each cell storing a single value. Raster data requires more storage space than vector data as the representation cannot automatically assume the inclusion of intermediate points given the start and end points. In vector data, geographic features are represented as geometrical shapes such as points, lines, and polygons. Vector data is represented in GIS using different file formats such as GML and shapefile, among others. In our experiments, we use GML file format (XML-based) to represent input and output GIS data.

For identification of overlaying-map-features, different algorithms based on uniform grid, plane sweep, Quad-tree, and R-Tree have been proposed and implemented on classic parallel architectures [81–83]. Franklin et al. [29] presented the uniform grid technique for parallel edge-intersection detection. Their implementation was done using Sun 4/280 workstation and 16 processor Sequent Balance 21000. Waught et al. [84] presented a complete algorithm for polygon overlay and the implementation was done on Meiko Computing Surface, containing T800 transputer processors using Occam programming language. Data partitioning in [29, 84–86] is done at spatial level by superimposing a uniform grid over the input map layers. Armstrong et al. [87] presented domain decomposition for parallel processing of spatial problems. While a wealth of research shows gains in performance over sequential techniques [88, 89], its application in mainstream GIS software has been limited [90, 91]. There has been very little research in high volume vector spatial analysis [92] and the existing literature lacks an end-to-end parallel overlay solution.

5.2.2 R-Tree

R-Tree is an efficient spatial data structure for rectangular indexing of multi-dimensional data; it performs well even with non-uniform data. R-Tree data structure provides standard

functions to insert and search polygons by their bounding box co-ordinates. We use R-Tree for intersection detection among polygons required for efficient overlay processing. Searching for overlap in R-tree is typically an $O(log_m n)$ operation where n is the number of nodes and m is the number of entries in a node, although it can result in $O(n)$ complexity in the worst case. We use Guttman's algorithm [93] for R-Tree construction and search operation.

5.2.3 Crayons system on Azure cloud

We have earlier developed a distributed cloud-based framework named Crayons [94] to execute traditional polygon overlay analysis algorithms on Windows Azure cloud platform. Windows Azure platform is a computing and service platform hosted in Microsoft data centres. Its programming paradigm employs two types of processes called web role and worker role for computation. For communication between web role and worker roles, it provides queue-based messaging and for storage it provides blobs and tables. Three load balancing mechanisms showing excellent speedup in Crayons are discussed in the paper [94].

However, Windows Azure platform lacks support for traditional software infrastructures such as MPI and map-reduce. Our current work generalizes Crayons by porting it to a Linux cluster with support for MPI framework. Porting a cloud application has its own set of challenges and we discuss some of them in this paper. We also made improvements on top of the Crayons' design by incorporating R-Tree - an efficient spatial data structure. We firmly believe that cluster based faster and efficient end-to-end GIS solution will aid GIS community as a whole.

5.2.4 Clipper Library

For computing map overlay over a pair of polygons, we use the GPC library which is an implementation of polygon overlay methods as described in [64]. The GPC library handles polygons that are convex or concave and self-intersecting. It also supports polygons with holes or polygons comprising of several disjoint contours. The usage of this widely used library shows the interoperability and accuracy of our approach for our polygon overlay

solution. GPC library supports intersection, union, difference, and X-OR. The output may take the form of polygon outlines or tristrips. We analyze and report *intersection* overlay operation throughout this project since it is the most widely used and is a representative operation. Nevertheless, our system can be extended to other operations as well without any change.

5.3 Master-Slave architecture based system design

We have implemented two versions of our system employing static and dynamic load balancing. The two versions further differ in the way task creation is carried out. Our system has a four-step workflow which consists of input file parsing, task creation, overlay computation, and output file creation.

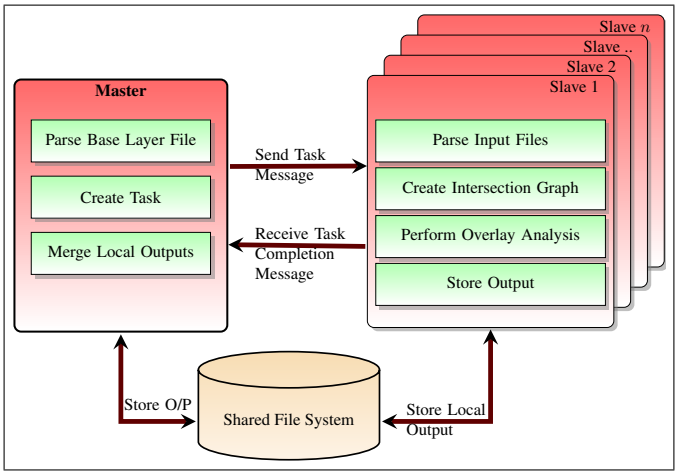

Figure (5.2) *Architecture with dynamic load balancing.*

5.3.1 Architecture with dynamic load balancing using R-tree

Figure 5.2 shows master-slave architectural diagram for dynamic load balancing version with task creation by employing R-Tree. The first step of our workflow starts with both the master and the slave processes parsing the GML files - only base layer for master process

and both the base and overlay layer files for slave processes. This redundancy later helps in task creation and processing.

Once parsing is done, each slave process builds its R-Tree using the bounding boxes of the overlay layer polygons. Master process determines the total number of base layer polygons and dynamically partitions those polygons into small chunks. The start and end indices of each chunk are sent to the slave processes. Once a slave process receives the polygon indices, the next step is to search its local R-Tree for potentially intersecting overlay layer polygons. Each slave process creates an intersection graph where each polygon from the base layer is connected with all of the polygons from overlay layer that can potentially intersect with it. Algorithm 4 describes the steps involved in creating the intersection graph based on R-Tree. Once the intersection graph has been created, each polygon from base layer and the connected polygons from overlay layer are stored together as a primitive overlay task for a slave process for processing. The overlay processing itself is carried out by invoking the built-in overlay function of the GPC library. The output from this library is a polygon structure that is converted to its equivalent GML representation. Once a slave worker finishes processing its tasks, it sends a message to the master worker, who in turn sends indices of the next chunk to be processed. Finally, when all tasks are completed, the master process merges all the local output files to generate the final output GML file.

Algorithm 4 R-Tree based algorithm to create intersection graph

INPUT: Set of Base Layer polygons S_b and Set of Overlay Layer polygons S_o
OUTPUT: Intersection Graph (V,E), where V is set of polygons and E is the set of edges among polygons with intersecting bounding boxes.

```
Create an R-Tree R using the bounding boxes of S_o
for all base polygon B_i in set S_b do
  for each polygon O_j in R-Tree R with bounding box intersecting with that of B_i do do
    create an edge (B_i, O_j) in graph G
  end for
end for
```

We experimented with different grain (chunk) sizes guided by the distribution of poly-

gons in a map layer. For non-uniform load, grain size should be smaller to account for load imbalance in comparison to the uniformly distributed data. The master process can be visualized as the owner of a pool of tasks from which it assigns tasks to slave processes. The slave processes continuously check with the master process for new tasks once they are done processing their individual tasks. The master-slave communication is message-based, handled by MPI send/receive primitives. The message size is intentionally kept small to lower the communication overhead while the message count (and thus the grain size) is determined by empirical data. The drawback of this version is the overhead of duplicate R-Tree creation at all of the slave processes.

5.3.2 Architecture with dynamic load balancing using sorting-based algorithm

We used sequential R-Trees for creating intersection graphs in the previous version. The only problem here is that it does not scale well since every slave process is building its own complete R-Tree redundantly. Therefore, we used a different algorithm for creating intersection graph, which is based on sorting the polygons on their bounding boxes [83]. In this algorithm, polygons are traversed to create an intersection graph where each polygon from the base layer is connected with all of the polygons from overlay layer that can potentially intersect with it. Algorithm 5 has the details of the sorting-based algorithm for detecting polygon intersection. In this algorithm, the slave processes perform graph creation only for their portion of base layer polygons. The time complexity of this algorithm is very high compared to R-Tree based search but for very large number of processors (more than 80), it will outperform R-Tree based approach since it avoids redundant R-Tree creation. As we will see in the results section (Section 6.4), usage of R-Tree is desirable when there are a few nodes in the cluster. The key intention in developing the two versions was to cut down the time taken for intersection graph construction and task creation.

Algorithm 5 Sorting-based algorithm to create intersection graph

INPUT: Set of Base Layer polygons S_b and Set of Overlay Layer polygons S_o
OUTPUT: Intersection Graph (V,E), where V is set of polygons and E is edges among polygons with intersecting bounding boxes.

```
Quicksort set S_o of overlay polygons based on X co-ordinates of bounding boxes
for all base polygon B_i in set S_b of base polygons do
  find S_x ⊂ S_o such that B_i intersects with all polygons in set S_x over X co-ordinate
  (binary search over S_o)
  for all overlay polygon O_j in S_x do
    if B_i intersects O_j over Y co-ordinate then
      Create Link between O_j and B_i
    end if
  end for
end for
```

5.3.3 Our system with Static Load Balancing

Based on the method of intersection graph creation, we have two flavors that employ static load balancing. One of them makes use of R-Tree (Algorithm 4) and the second one simply uses binary search on polygons sorted on bounded boxes (Algorithm 5). The rationale behind developing static version is to assess the communication cost between master and slave processes involved in the dynamic version. Although the basic workflow is similar to dynamic version, task partitioning in the static version is straightforward. After independently parsing both of the input GML files, the slave processes equally divide the base layer among themselves based on their process-IDs and perform task creation only for their portion of base layer polygons, thereby obviating any need for master-slave communication. Master process is only responsible for merging the files created by different slave processes.

Output file creation is initiated by master process once all the slave processes finish their tasks and terminate. Termination detection differs in static and dynamic version and is handled by master process. In static version, once master process receives task completion messages from all slave processes, it merges the partial (local) output GML files created by respective slave process to yield final output and finally terminates. On the other hand, in the dynamic version, master process interactively tracks the completion of tasks and once

all tasks are finished, it sends termination message to each slave process and generates an output GML file.

5.3.4 MPI related Issues

As seen in section 5.3, each and every slave process reads input files. The parsing of files is a sequential bottleneck here and it is performed redundantly. This problem worsens, due to I/0 contention for the shared file system, when the number of processors increase. It is, therefore, intuitive to let only master process parse files once, create task and schedule them dynamically for slave processes to execute.

1. MPI Send/Receive issue: In order to distribute work among slave processes we need to communicate the tasks using MPI send/receive primitives. The vector data needed for the GPC library is a nested structure containing polygon information including number of contours, number of vertices, bounding box information etc. The polygonal data is non-contiguous data of mixed data types. Even though MPI provides derived data type for user-defined data structures and packing routines, these do not work for dynamically allocated nested structure that we had to use.

2. Cost of serialization: To avoid the MPI Send/Receive issue mentioned above, we serialized the polygonal data as a text stream. In this version, master process creates tasks, serializes and sends to slave processes. Each slave process receives the message and deserializes to get the task and performs overlay computation. Vector data tends to be large in size and experimentally we found that the cost of serialization/deserialization and message communication is huge even for smaller grain sizes.

5.3.5 Clipper Library Related Issues

GPC library is a well-known open-source library for basic polygon clipping operations but it has some limitations. First and foremost, the GPC library only supports four operations - intersection, X-OR, union, and difference. It does not support *Equals*, *Crosses*,

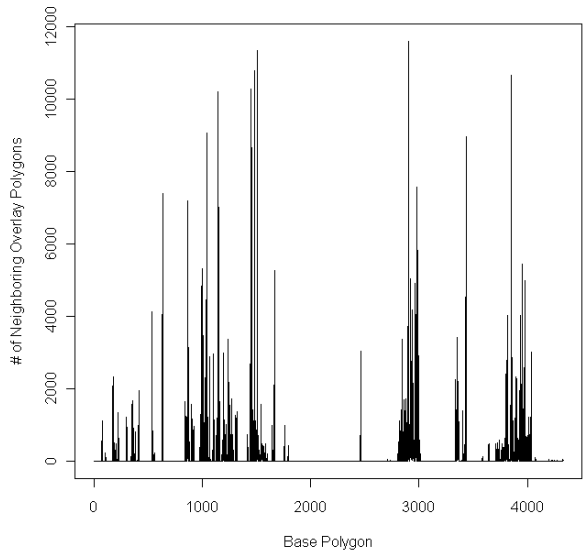

Figure (5.3) *Skewed load distribution for smaller data set.*

Within, *Contains*, *Disjoint*, *Touches*, and *Overlap*. Moreover, the library does not preserve the relationship between a hole and the polygon that contains this hole.

5.3.6 Timing Characteristics and Experiments

We have performed our experiments on a Linux cluster that has 80 cores distributed among 9 compute nodes. The cluster contains 1) four nodes with each having two AMD Quad Core Opteron model 2376 (2.3 GHz), 2) one node with four AMD Quad Core Opteron model 8350 (2.0 GHz), and 3) four nodes with each having two Intel Xeon Quad Core 5410 (2.33 GHz) connected using Infiniband switch. In our cluster all the nodes share the same file system hosted at the head node.

We experimented with two different sets of data. First set consists of files of size 770 MBs containing 465,940 polygons and 64 MBs containing 8600 polygons. This data set has skewed load distribution. The second data set consists of files of size 484 MBs containing 200,000 polygons and 636 MBs containing 250,000 polygons. This is the larger data set but the load distribution is uniform here. Figure 5.3 and Figure 5.4 shows the load distribution plots

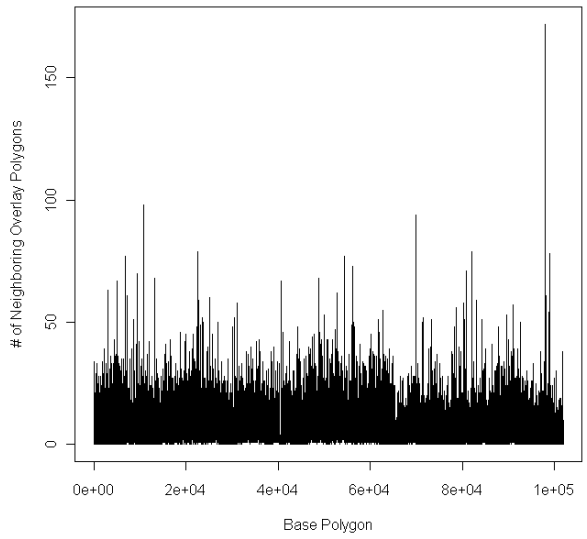

Figure (5.4) *Comparatively uniform load distribution for larger data set.*

for a sample of the base layer polygons used in experiments. To calculate absolute speedup against the one-processor time without any parallel overhead, we used the execution time of R-Tree based version on a single processor. Unless otherwise stated, all benchmarking has been performed over the end-to-end 1-processor time (the process of taking two GML files as input, performing overlay processing, and saving the output as a GML file) using R-Tree based algorithm on the smaller data set.

Figure 5.5 shows the absolute speedup when we use sorting-based algorithm. For dynamic version using R-tree, the overall end-to-end (starting from two input GML files to producing output GML file) acceleration is about 15x as shown in Figure 5.6. R-Tree based version shows better performance in comparison to the sorting based version. Dynamic version works better than static version due to the non-uniform distribution of polygonal data as we mentioned earlier. Moreover, master-slave communication time is very small owing to the small size of communication messages.

Figure 5.8 and 5.9 shows the execution time breakdown of subprocesses for the static versions. Task creation step involves data partitioning so that slave processors can work on

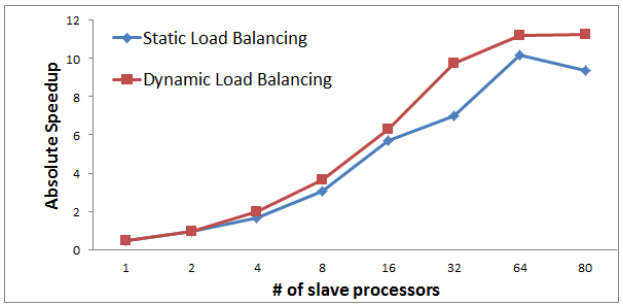

Figure (5.5) *Performance impact of varying worker processors using sorting-based algorithm.*

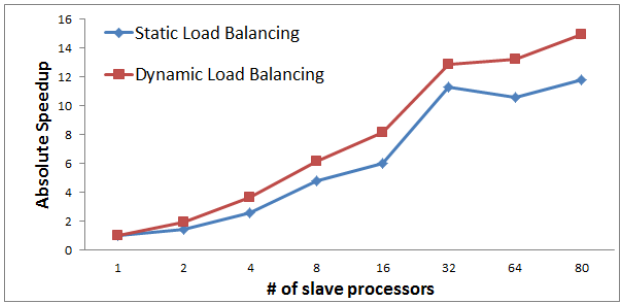

Figure (5.6) *Performance impact of varying worker processors using R-Tree based algorithm.*

independent tasks. Overlay processing step involves computing overlay and writing the local output polygons to GML files. Figure 5.10 and 5.11 shows the execution time breakdown of subprocesses for dynamic versions. The reported time in the above-mentioned figures is the average time recorded by noting the time for each of the three subprocesses, i.e., parsing, task creation, and overlay task processing, for each slave process and then taking an average. Overlay task processing includes assigning the tasks to *GPC* library and is followed by the output storing step where the local outputs are stored in the shared file system as a separate file (one file for each slave process). The overlay processing time in case of R-Tree based version is more than sorting-based version for the same dataset as can be seen from Figure 5.9 and Figure 5.11. This is due to the fact that when we use R-Tree datastructure, we get more potentially intersecting polygons for a given base layer polygon in comparison to the sorting-based version. It should be noted here that all the potentially intersecting

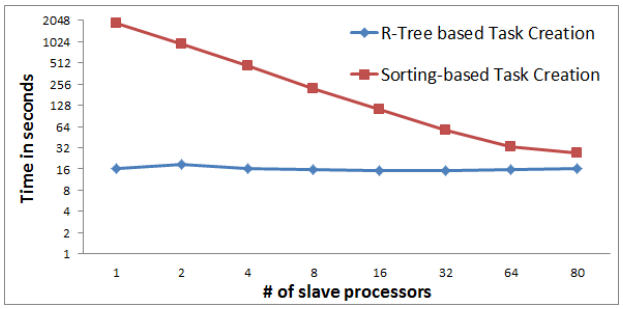

Figure (5.7) *Performance impact of varying worker processors on task creation.*

polygons may not actually intersect. So, this does not affect the correctness of our final output.

Figure 5.7 shows that the task creation using sorting-based algorithm takes much more time in comparison to the R-Tree based algorithm. For a cluster with more than 80 processors, the sorting-based algorithm may take lesser time than R-Tree based algorithm. Although the time taken for task creation step in case of sorting-basaed algorithm decreases along with increase in number of slave processors, it is not suitable for a cluster with a few number of nodes. The comparison shows efficiency of R-Tree based algorithm over sorting-based algorithm for intersection graph creation. Use of R-Tree makes sure that the percentage of sequential portion in the overall algorithm is reduced thereby increasing the parallelism by Amdahl's law. This is the main reason for the improvement over the sorting-based algorithm. Thus, for cluster with few number of nodes, R-Tree based algorithm works better.

For both the smaller and the larger data sets, the overlay processing task scales very well for static as well as dynamic loadbalancing as the number of slave processors increase. Since, the size of the message is very small, the communication cost is small in comparison with the cost of input file parsing and task creation. However, we observed that due to the contention for parallel file access, parsing of the input GML files and writing of output files takes longer with the number of processors growing. Further scaling of this system is challenging unless high throughput file access can be supported.

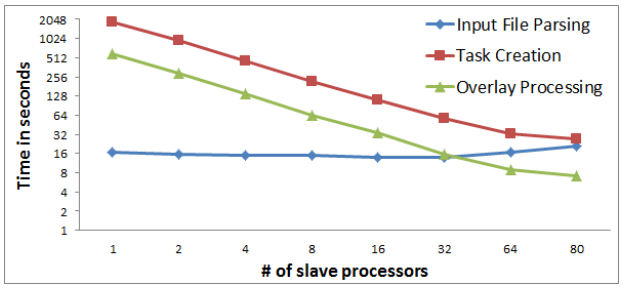

Figure (5.8) *Execution time breakdown for static version (Sorting-based algorithm).*

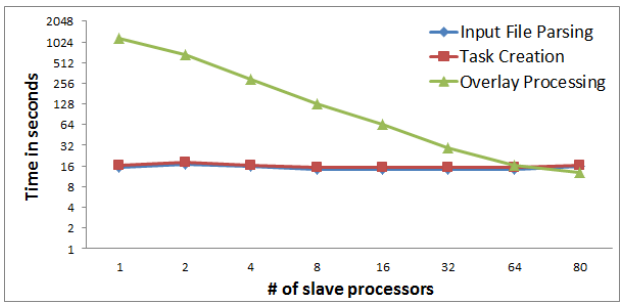

Figure (5.9) *Execution time breakdown for static version (smaller data set) (R-Tree based algorithm).*

5.4 Optimizations based on file and space partitioning

In the previous section, we discussed how to perform polygon overlay without spatial and file partitioning. The design was based on master/slave architecture. The implementation required parsing of entire input files and sequential R-tree construction by all the slave nodes. In this section, we will also discuss how to partition input polygon files using uniform grid, distribute the polygons, and coordinate work among multiple compute nodes using message passing model (MPI).

In order to overlay two polygon layers, the first step is to find overlapping pair of polygons (e.g. $< p_i, q_i >$) from these two layers (filtering step) and then apply polygon clipping on these pairs. The individual pairs can be processed using an optimized sequential algorithm due to typically large number of such independent tasks to keep the cores busy.

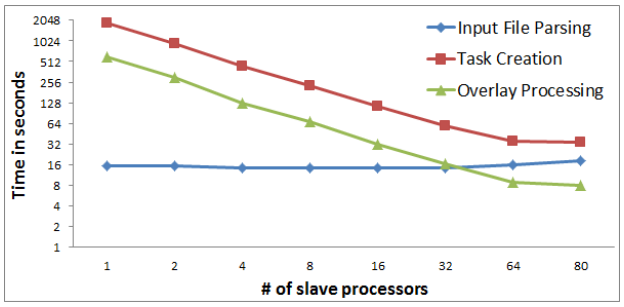

Figure (5.10) *Dynamic Load Balancing (Sorting-based algorithm).*

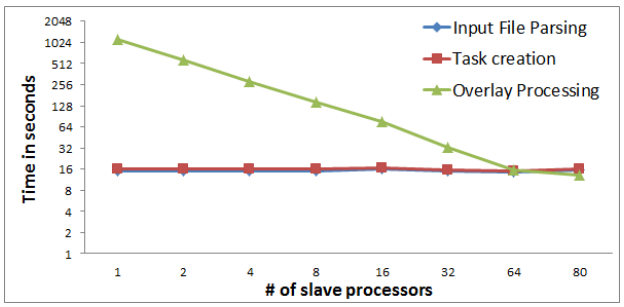

Figure (5.11) *Execution time breakdown for dynamic versions (smaller data set).*

Although not integrated yet into the *MPI-GIS* system, for further scalability, one may employ multiple cores and employ parallel Greiner-Horman and/or Parallel Vatti's algorithm that we have developed. The filtering step can be done by building an R-tree using Minimum Bounding Rectangles (MBR) of one polygon layer and querying the R-tree [95] using MBR of polygons from the other layer. Now, in the following subsections, we will discuss input data processing, spatial partitioning, and communication aspects of *MPI-GIS*.

5.4.1 Input Data Processing and Organization

The *MPI-GIS* system assumes that GIS data is stored in the Shapefile format [96], and every computing node maintains a local copy of entire GIS data in its local drive. For polygon overlay processing, two Shapefiles are required: one for subject polygons and the other for clip polygons. While the term "shapefile" is widely used, a shapefile actually contain a set

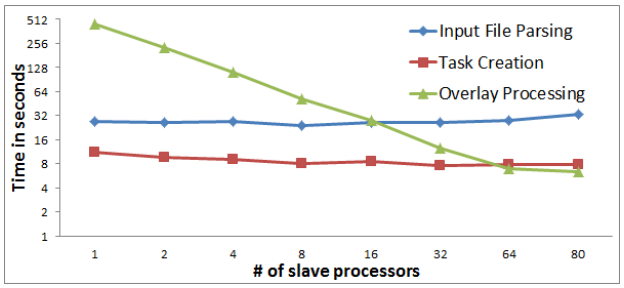

Figure (5.12) *Static Load Balancing (R-Tree based algorithm).*

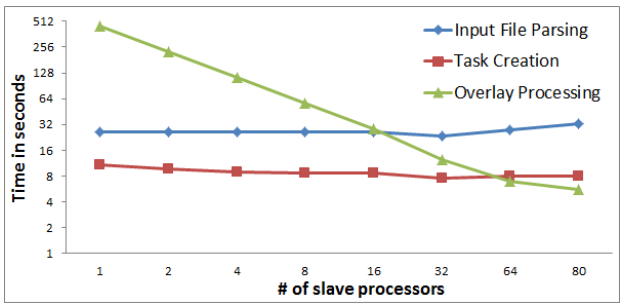

Figure (5.13) *Dynamic Load Balancing (R-Tree based algorithm).*

of files: three mandatory files and other optional files. The three mandatory files are a main file (.shp), an index file (.shx) and a bBASE table (.dbf). The main file is a direct access, variable-record-length file in which each record describes a shape with a list of its vertices. In the index file, each record contains the offset of the corresponding main file record from the beginning of the main file. The dBASE table contains feature attributes with one record per feature [96]. For large shapefiles, the index file is much smaller than the main file. By reading the index file into main memory, a computing node is able to read any shape record in the main file in a random-access fashion. This thus enables *MPI-GIS* to parallelize I/O operations with each computing node reading only a portion of GIS data.

5.4.2 Spatial Partitioning

First, computing nodes collaborate to read the entire polygon MBRs from disks with input shapefiles pre-distributed across the local hard drives of all compute nodes. Each node

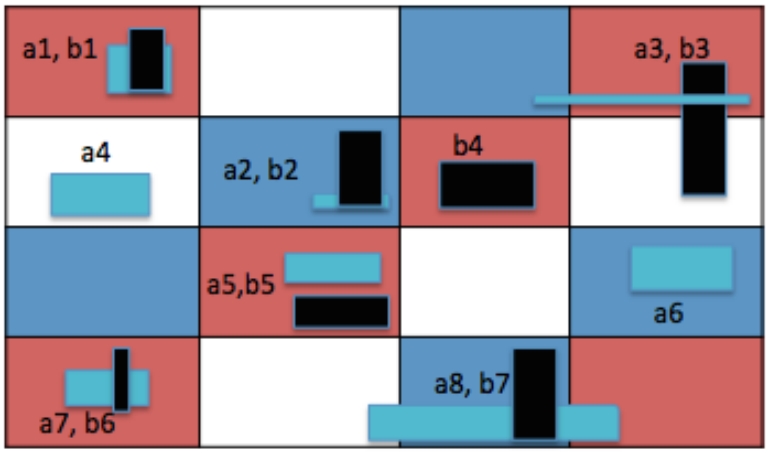

Figure (5.14) *Spatial distribution of polygonal MBRs from subject layer* $A = \{a_1, a_2, .., a_8\}$ *and clip layer* $B = \{b_1, b_2, .., b_7\}$ *(shown in black) in a uniform grid. Three processors P1 (red) , P2 (white) and P3 (blue) are assigned to grid cells in a round-robin fashion. These processors have local access to only a subset of polygons from layer A and B as shown: P1 has access to A1 =* $\{a1, a2, a3\}$ *and B1 =* $\{b5, b6, b7\}$*. P2 has access to A2 =* $\{a7, a8\}$ *and B2 =* $\{b1, b2\}$*. P3 has access to A3 =* $\{a4, a5, a6\}$ *and B3 =* $\{b3, b4, b5\}$*.*

only reads a portion of MBRs locally. Based on the X and Y coordinates, each MBR is mapped to a cell in a 2-dimensional uniform grid. These grid cells and the MBRs in them are assigned to computing nodes for processing in a round robin fashion, or alternatively, proportional to the compute capability of the node, thus defining cell ownership. Since MBRs belonging to a grid cell may not be locally available to the cell owner, an all-to-all exchange among computing nodes is employed to obtain its share of polygon IDs and MBRs. Computing nodes then read vertices of polygons in their assigned cells using polygon IDs (fast access provided by shapefile indexing), and organize these MBRs and vertices as subject polygon objects and clip polygon objects. The MBRs of subject polygons at each grid cell are organized into R-trees. A batch of queries using the MBRs in these clip polygon objects are carried out on the R-tree to produce tasks, each comprising a pair of potentially intersecting subject polygon and clip polygon. These are subsequently processed using Clipper library [97].

Initially, the dimension of the universe is determined which is a minimum bounding box spatially containing all the polygons from the base layer and the overlay layer. Then the

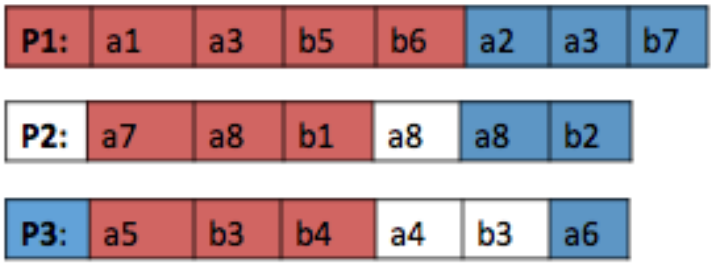

Figure (5.15) *Each processor arranges the MBRs for P1, P2, and P3 in order (color coded) in All-to-All Send Buffer.*

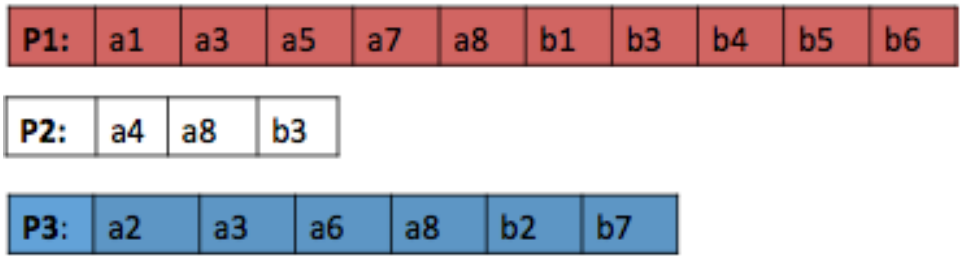

Figure (5.16) *MPI Communication of polygonal MBRs distributed in uniform grid among three processors.*

dimension of grid cells is computed based on the number of partitions required. In order to ensure proper load-balancing, the universe is divided into N uniform sized cells where N>>P (number of processors) and these cells are assigned to processors in a round-robin fashion. Polygons may span more than one grid cell. This may result in redundant polygon clipping of a pair of polygons across different cells. To avoid such redundant computations, we employ the reference-point duplicate avoidance technique [98].

5.4.3 MPI Communication

A polygon may belong to one or more cells and since we know the bounding boxes of all the grid cells, each process can determine to which cell(s) a locally-read polygon belongs to. Figure 5.16 a) shows the distribution of MBRs of polygons in a uniform grid. The polygon-to-processor mapping is cell-based and it is carried out in two phases. Initially, each MPI

process reads chunk of subject and clip layer polygons, assigns the polygonal MBRs to the overlapping cells and prepares a communication buffer for an all-to-all collective communication (MPI_Alltoallv). To illustrate communication among processors with MPI, Figure 5.16 also shows the communication buffers filled with MBRs. After this global communication phase, each compute node gets all the polygonal MBRs and their IDs for its allocated cells from both map layers, and reads the polygons (vertices) of all its cells from its local drive.

5.4.4 MPI-GIS Overlay Processing System Evaluation

Hardware Platform: We used *NERSC's Carver* cluster [72] which is an IBM iDataPlex cluster with each node having two quad-core Intel Xeon X5550 (Nehalem) 2.67 GHz processors (eight cores/node). All nodes are interconnected by 4X QDR InfiniBand interconnect, providing 32 Gb/s of point-to-point bandwidth for high-performance message passing. We used the Open MPI library and C programming language implementation.

Spatial Data: As real-world spatial data, we selected polygonal data from Geographic Information System (GIS) domain. We experimented with shapefiles from *http://www.naturalearthdata.com* and *http://resources.arcgis.com*. The details of the datasets are provided in Table 6.2. Datasets 3 and 4 are spatial features taken from telecommunication domain [67].

Table (5.1) *Description of real-world datasets.*

	Dataset	**Polygons**	**Edges**	**Size(MBs)**
1	ne_10m_urban_areas	11,878	1,153,348	20
2	ne_10m_states_provinces	4,647	1,332,830	50
3	GA_telecom_base	101,860	4,488,080	150
4	GA_telecom_overlay	128,682	6,262,858	200
5	USA_Water_Bodies	463,591	24,201,984	520
6	USA_Block_Boundaries	219,831	60,046,917	1300

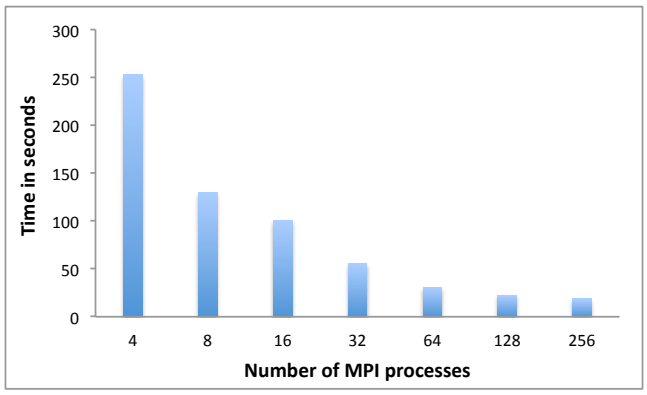

Figure (5.17) *Execution time of MPI-GIS with varying number of MPI processes for Intersect (#5, #6) on Carver cluster with 32 compute nodes having 8 cores/node.*

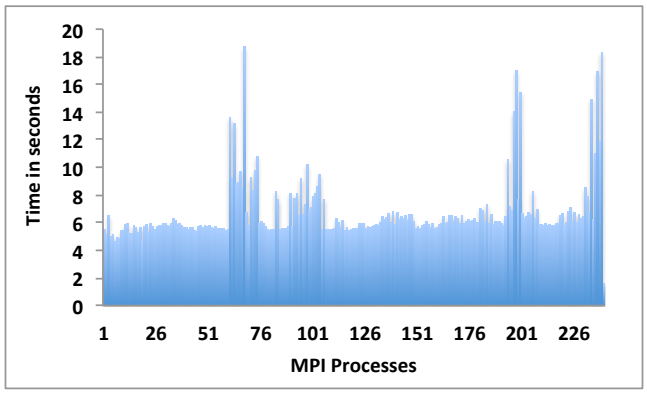

Figure (5.18) *Time taken by different MPI processes (256 processes in total) in MPI-GIS for Intersect (#5, #6) on Carver cluster using 32 compute nodes, each having 8 cores.*

To obtain some baseline performance, we compared *MPI-GIS* against ArcGIS 10.1 which is a commonly used GIS software tool. We ran ArcGIS on Windows 7 having 2.4 GHz Intel Core i5 processor and 4 GB memory. We developed a sequential overlay processing system using R-tree filtering and sequential polygon clipping library. We ran our sequential system on Intel Xeon (2.67 GHz). We applied geometric union and geometric intersection operations on three pairs of layers taken from real-world datasets. As we can see in the Figure 5.20, ArcGIS performed better than our sequential system for datasets in Table 6.2.

Figure 5.17 shows the scalability of *MPI-GIS* running on 32 nodes of *Carver* for the

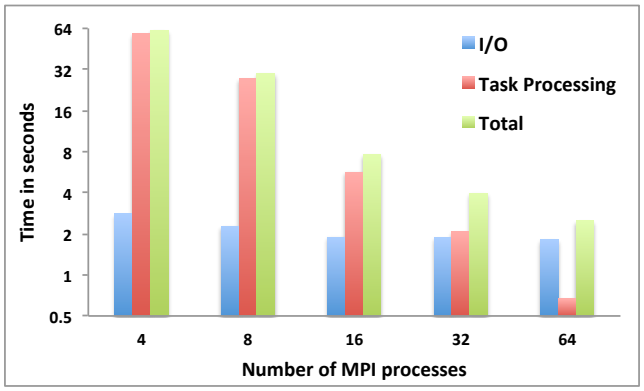

Figure (5.19) *Execution timing breakdown in MPI-GIS system for Intersect (#3, #4) on Carver cluster with 8 compute nodes having 8 cores/node.*

largest size data (number 5 and 6 in Table 6.2). Figure 5.18 shows the time taken by each MPI process. As we can see in this figure, some processes are three times slower than other processes due to load imbalance which affects further scalability. Figure 5.19 shows the execution profile for different components in *MPI-GIS* for the medium size data on a cluster of 8 nodes. Task creation takes a very small portion of the total execution time. Task processing is the most time-consuming component in the system. However, task processing time is greatly reduced as the number of computing nodes increases, showing promise for further scalability. File I/O remains a bottleneck due to I/O contention among MPI tasks in a given node. Figure 5.20 shows the execution time taken by *MPI-GIS* in comparison with ArcGIS and sequential version for real world datasets for union and intersection operations. ArcGIS overlay computation is faster than our sequential version for all the datasets. *MPI-GIS* performs efficiently on a cluster of 32 nodes yielding 25x to 50x speedup when compared to ArcGIS.

5.4.5 Comparison with earlier work

Our multi-threaded implementation described in [69] was based on partitioning of spatial data into horizontal slabs. The threads were allocated to consecutive slabs. For Intersect (#1, #2), we got about 3x speedup using a single compute node in comparison to ArcGIS.

Software/Library	Intersect(1,2)	Intersect(3,4)	Intersect(5,6)	Union(1,2)	Union(3,4)	Union(5,6)
ArcGIS	17	110	830	21	135	1147
Sequential	43	121	2297	63	143	3109
MPI-GIS	0.68	2.5	19	0.89	2.7	28

Figure (5.20) *Performance (seconds) of commonly used software application tool/libraries versus MPI-GIS using real-world datasets on Carver cluster with 32 compute nodes, each having 8 cores.*

Using *MPI-GIS*, we are getting about 25x speedup. Now, we are able to use more than one compute node. Moreover, for load balancing, we are using round-robin scheduling.

Our earlier MPI based implementation required parsing of entire input files and sequential R-tree construction by all the slave nodes [17, 71]. It takes about 5 seconds to read third and fourth datasets and construct R-tree for Intersect(#3, #4). It takes about 17 seconds to read fifth and sixth datasets and construct R-tree for Intersect(#5, #6). As a result, its scalability was limited upto 8 nodes and its speedup was only 15x when compared to sequential version. In *MPI-GIS*, file reading is divided among worker nodes and spatial partitioning improves data locality. Moreover, instead of creating one large R-tree, in *MPI-GIS*, smaller R-trees are constructed in grid cells which also improves R-tree construction and query performance. From our previous experiments using MPI [71] and Hadoop [16] for overlaying 2 GB (medium-size) of polygonal data, running on a cluster with 10 nodes, we found MPI-based system to perform better than Hadoop-based system owing to higher overhead of Hadoop. Parallel polygon overlay implementation of Franklin's uniform grid [29] using a single CPU and a GPU is presented in [99].

PART 6

MAPREDUCE ALGORITHMS FOR GIS POLYGONAL OVERLAY PROCESSING

6.1 Introduction

Computational geometry algorithms are used in different fields. One of the complex geometric algorithms used in GIS setting is polygonal overlay, which is the process of superimposing two or more polygon layers to produce a new polygon layer. In GIS, overlay operation is a key analysis function used in combining the spatial and attribute data of two input map layers. Polygon overlay is a type of map overlay but on GIS polygonal (also known as "vector") data where the spatial input data is represented in the form of points, lines and polygons and it is much more complex than raster data computation.

In some cases when large volumes of spatial data is deployed for overlay analyis, it becomes a time consuming task, which sometimes is also time sensitive. GIS scientists use desktop based sequential GIS systems for overlay tasks. The desktop GIS software typically takes hours to perform overlay for large data sets which makes it useless for real time policy decisions. For emergency response in the US, for example, disaster-based consequence modeling is predominantly performed using HAZUS-MH, a FEMA-developed application that integrates current scientific and engineering disaster modeling knowledge with inventory data in a GIS framework. Depending on the extent of the hazard coverage, data sets used by HAZUS-MH have the potential to exceed the capacity of standard desktops for detailed analysis, and it may take several hours to obtain the results. As such, large scale overlay processing can potentially benefit from MapReduce based implementation on a cluster or cloud.

In this paper, we present the adaptation and implementation of a polygon overlay algorithm (Algorithm 6), which originates from the area of Geographical Information Systems.

We describe a system to execute distributed version of this algorithm on a Linux cluster using Hadoop MapReduce framework. Parallelizing polygonal overlay with MapReduce has its own set of challenges. First, MapReduce focuses mainly on processing homogeneous data sets, while polygon overlay has to deal with two heterogeneous data sets. Second, the polygon overlay operations are complex and time-consuming, and vector objects are generally larger and more complex than words or URL strings in common MapReduce applications. Third, partitioning vector data evenly to yield good load balance is non-trivial. Our parallel system is based on R-tree data structure which works well with non-uniform data. Our system carries out end-to-end overlay computation starting from two input GML (Geography Markup Language) files, including their parsing, employing the bounding boxes of potentially overlapping polygons to determine the basic overlay tasks, partitioning the tasks among processes, and melding the resulting polygons to produce the output GML file. We describe the software architecture of our system to execute the algorithm and discuss the design choices and issues. We analyze and report *Intersection* overlay operation in our paper since it is the most widely used and representative operation. Nevertheless, our system can be extended to other operations as well without any change. Our rigorous instrumentation of the timing characteristics of various phases of the algorithm point out portions of the algorithm which are easily amenable to scalable speedups and some others which are not. We have experimented with two data sets and achieved up to 22x speedup with dataset 1 using 64 CPU cores and up to 8x with dataset 2 using 72 CPU cores.

We have implemented overlay algorithms in MapReduce in three different forms, namely, i) with a single map and reduce phase, ii) with chaining of MapReduce jobs, and iii) with a single map phase only. We also show performance comparison among these three different versions. In the literature, we found bottom-up construction of R-tree by bulk loading strategies implemented using MapReduce. In this paper, we present a top-down construction of R-tree in a distributed setting using a grid based approach. Our specific technical contributions include

- porting an MPI based spatial overlay system [100] to Hadoop MapReduce platform,

- an overlay algorithm with chaining of MapReduce jobs,
- a grid based overlay algorithm with a single map and reduce phase, and
- an overlay algorithm with map phase only using *DistributedCache*.

The rest of this paper is organized as follows: Section 6.2 reviews the literature briefly and provides background on GIS data, R-tree data structure, general polygon clipper (GPC) library, and MapReduce framework, and expresses polygon overlay as a graph problem. Section 6.3 describes four (including a naive algorithm) MapReduce based algorithms. Our experimental results and other experiments are in 6.4.

6.2 Background and Literature

6.2.1 Map Overlay

Map Overlay is one of the key spatial operations in GIS. It is the process of interrelating several spatial features (points, lines, or polygons) from multiple data sets, which creates a new output vector dataset. For instance, one map of United Sates representing population distribution and another map representing the area affected by hurricane Sandy can be overlaid to answers queries such as "What is the optimal location for a rescue shelter?". Clearly, it is often needed to combine two or more maps according to logical rules called overlay operations in GIS. An Intersection overlay operation, similarly, defines the overlapping area and retains a set of attribute fields for each. It should be noted here that the process of map overlay in case of raster data is entirely different from that of vector data and our solution deals with vector data. Since resulting topological features are created from two or more existing features of the input map layers, the overlay processing task can be time consuming.

For computing basic overlay of a pair of polygons, we use the General Polygon Clipper (GPC) library which is an implementation of polygon overlay methods [64]. The input to this library is a pair of polygons and an overlay operator like Intersection, Union, Difference, and XOR. The GPC library handles polygons that are convex or concave and self-intersecting. It

also supports polygons with holes. The usage of this widely used library leads to accuracy of our approach for our polygon overlay solution. Algorithm 6 presents the high level sequential algorithm for polygon overlay.

Algorithm 6 Sequential Polygon Overlay algorithm [100]

INPUT: Set of Base layer polygon A and set of overlay layer polygons B
OUTPUT: Set of Ouptut polygons O
for all polygon p_A in A **do**
  Step 1: find all intersecting polygons I from set B
  Step 2: for each p_B in I
  Step 3: compute p_O = overlay (p_A, p_B)
  Step 4: add polygon p_O to O
end for

When we perform overlay between two map layers, a polygon from base layer A can spatially overlap with zero or more polygon(s) from overlay layer B. This overlapping relationship between polygons from two different layers can be represented in the form of an intersection graph. This can be defined as a graph $G = (V, E)$ where V is set of polygons, $V = \{A \bigcup B\}$ and E is the set of edges between polygons such that $\forall$ u,$v \in V$, $u \in A$ and $v \in B$, there exists an edge (u, v) in G if and only if u overlaps with v.

6.2.2 R-tree Data Structure

R-tree is an efficient spatial data structure for rectangular indexing of multi-dimensional data. R-tree data structure provides standard functions to insert and search spatial objects by their bounding box co-ordinates. Searching for overlap in R-tree is typically an $O(mlog_m n)$ operation where n is the total number of nodes and m is the number of entries in a node. However, it can result in $O(n)$ complexity in the worst case. We use Guttman's algorithm [93] for R-tree construction and search operations. R-tree can be constructed in top-down fashion or by using bulk-loading bottom-up techniques.

6.2.3 MapReduce Framework

The cloud computing framework based on Hadoop MapReduce provides a promising solution to solve the problems involving large-scale data sets. The input data is viewed as a stream of records comprising of key-value pairs. As the name suggests, in MapReduce there are two phases namely map phase and reduce phase. A map phase consists of map tasks where each map task can run in parallel and works on a portion of the input. A map task produces intermediate key-value pairs which are processed in parallel by reduce tasks in reduce phase. The reduce phase of a job cannot begin until the map phase ends. A shuffle and sort phase is implicitly carried out before reduce phase begins. Under the MapReduce programming model, a developer needs to provide implementations of the mapper and reducer. In addition to *map* and *reduce* methods, Hadoop MapReduce framework provides *initialize* or *setup* and *cleanup* methods which can be overridden by application developers. *Initialize* is called once at the beginning of the map task and *cleanup* is called at the end of the map task. The framework takes care of task scheduling, data locality and re-execution of the failed tasks. Google's MapReduce and Apache Hadoop are two popular implementations of MapReduce. In Hadoop, both the input and the output of the job are stored in a file-system known as Hadoop Distributed File System (HDFS). MapReduce framework provides a facility known as *DistributedCache* to distribute large, read-only files. The framework copies the files onto the slave nodes before any tasks for the job are executed on that node. Hadoop provides -files or -archive directive to send the input file to all the slave machines while starting a job. In MapReduce, synchronization is achieved by the shuffle and sort barrier. Intermediate results generated are aggregated by reducers. After the shuffle and sort phase is over, all key-value pairs with same key are grouped together at one reducer. Combiners perform local aggregation, and partitioners determine to which reducer intermediate data are shuffled to.

6.2.4 Overlay and Miscellaneous Approaches

For identification of overlaying map-features, different algorithms based on uniform grid, plane sweep, Quad-tree, and R-tree have been proposed and implemented on classic parallel architectures [29, 82, 83, 101]. Edge intersection detection is a subproblem of computing polygon overlay and most of the research aims to parallelize edge intersection detection phase only. Franklin et al. [29] presented the uniform grid technique for parallel edge intersection detection. Their implementation was done using Sun 4/280 workstation and 16 processor Sequent Balance 21000. Waught et al. [84] presented a complete algorithm for polygon overlay and the implementation was done on Meiko Computing Surface, containing T800 transputer processors using Occam programming language. Data partitioning in [29, 84–86] is done by superimposing a uniform grid over the input map layers.

To the best of our knowledge, there are no papers on MapReduce based polygon overlay. Spatial join is similar to polygonal overlay and it is used for joining attributes from one feature to another based on the spatial relationship. MapReduce based algorithm for spatial join problem is discussed in [21]. Their strategies include strip based plane sweeping algorithm and tile-based spatial partitioning function. Tile based partitioning is used to partition the data into multiple uniform tiles and these tiles are assigned to map tasks in a round-robin fashion. Tile based partitioning may result in high overhead for replicating polygons across tiles for skewed data. Spatial analysis service system based on MapReduce programming model is proposed in [102]. A high level MapReduce algorithm for overlay analysis and R-tree index building for raster data is also presented but the paper does not include performance evaluation. The authors in [103] and [104] propose a method of bulk-loading spatial data for R-tree generation using the MapReduce framework. We presented a parallelization scheme for polygonal overlay based on Message Passing Interface (MPI) in [100]. In [105] and [94], we discuss static and dynamic load balancing strategies for parallelizing map overlay on Azure cloud platform. But these parallelization schemes are not applicaple to Hadoop-based MapReduce system since the latter lacks explicit message passing as in MPI framework (point to point send/receive) or the queues as in Azure for communication.

6.3 Map Reduce Algorithms

6.3.1 Problem Definition

The polygon overlay combines the input polygons from two different maps into a single new map. The input to binary map overlay are two map layers $L_1 = [p_1, p_2, .. , p_n]$ and $L_2 = [q_1, q_2, .. , q_m]$ where p_i and q_i are polygons represented as x,y co-ordinates of vertices. The output of the overlay operation is a third layer $L_3 = L_1 \times L_2 = [o_1, o_2, ... , o_k]$ represented by k output polygons and this output depends on the overlay operator denoted as $\times$. Overlay operators such as Union, Intersection, etc, determine how map layers are combined.

Our system has a four-step work-flow which consists of input file parsing, task creation, overlay computation, and output file creation. Here, we present four different algorithms using Hadoop MapReduce framework.

File partitioning in Hadoop distributed filesystem (HDFS) is based on input split size which can be 64 MB or 128 MB. Once split size is decided, Hadoop internally takes care of input file partitioning. The input in our application consists of GML files which contain polygon vertices and other spatial attributes spanning multiple lines bounded by XML tags. The straightforward partitioning of such a file based on split size may lead to one polygon spanning multiple input splits which will produce incorrect result. In order to ensure proper splitting of input data, we preprocess the input file such that a polygon is contained in a single line. Even then the splitting phase can result in one line getting split at an arbitrary position, but in this case Hadoop framework ensures that a line is completely contained in one input split only. A preprocessing MapReduce job can be executed to extract the polygon ID, Minimum Bounding Rectangle (MBR) and polygon vertices from each spatial object and these spatial attributes can be written in a single line for each object. Finally, the preprocessed input files can be sent to HDFS for further processing. This preprocessing phase is done offline.

6.3.2 Algorithm With Map and Reduce Phases

The input to the Algorithm 7 (our first parallel algorithm) is set of polygons from a base layer file and an overlay layer file which is stored in HDFS. Since there are two input files, a source tag is required to identify a polygon's parent data set. Every mapper receives a 64 MB file split from base layer or overlay layer as input. The input file split belongs to a single map layer and each mapper simply emits the polygon as key-value pair depending upon whether the polygon belongs to base layer or overlay layer. The term "EMIT" as used in MapReduce algorithms, means producing output as a key-value pair which gets written to HDFS. Since, the overlap relationship or intersection graph for polygons is not known at this stage, a brute force algorithm is applied here and the polygons are emitted in such a way so that in the reduce phase a base polygon is grouped with all overlay layer polygons for intersection detection. The actual overlay computation is thus delegated to reduce phase thereby ignoring local computation in the map phase and carrying forward all the polygonal data to the reduce phase. For the sake of clarity, we have intentionally omitted the key instead of writing a "null" as key while using "EMIT" for producing the final overlaid polygon as output.

Observation: The disadvantage of this algorithm is that (i) it produces too many key-value pairs and (ii) since intermediate key-value pairs are spilled to disks and thus involves writing to files, the overall performance degrades. This algorithm does not perform well because of the overhead of task creation and all-to-all communication during shuffle between Map and Reduce phases.

It is non-trivial to efficiently parallelize binary applications requiring two input sources in MapReduce framework. In map overlay which is a binary application, data locality may not be possible since default input file splitting may result in mapper tasks having a split from a single input source. In the absence of input chunks from both input sources in a mapper, overlay processing can not be done in the map phase itself. Instead, the data is simply forwarded to reduce phase for overlay processing which results in extra communication overhead. If a mapper possesses splits from both input sources, local overlay processing can

Algorithm 7 Naive MapReduce overlay algorithm

```
method MAP (id, polygon p)
{
extract source tag from p
if p is from base layer then
  EMIT: (id, p)
else
  for all base polygon id b do
    EMIT: (b, p)
  end for
end if
}

method REDUCE (base polygon id, [c1, c2, . . ])
{
divide [c1, c2, . . ] into set of overlay polygons C and a base polygon b
M ← get overlay polygons from C overlapping with b
for all c ∈ M do
  compute overlay (b, c)
  EMIT: output polygon
end for
}
```

be done but since it is possible that polygons in one mapper may overlap with polygons in other mappers, a chain of MapReduce jobs or more than one iterations are required to perform overall overlay in a distributed fashion.

6.3.3 Overlay Algorithm with Chained Map Reduce Phases

Here, in file preprocessing phase, we interleave the polygons from both base layer and overlay layer so that all the file splits get portions of input data from both the layers. In Algorithm 8, the overall work is divided into two distinct MapReduce jobs. The first job has only a map phase in which the bounding boxes of base layer polygons are emitted as output in order to aid in intersection detection carried out in the second MapReduce job. The output of the first job serves as input for the second job where all the mapper tasks create a local R-tree from the bounding boxes of the base layer polygons. The output of the first job consists of bounding boxes of all the base layer polygons which are copied to the *DistributedCache* before the second job starts. Now, these bounding boxes are accessible to all the mappers in the second job through the *DistributedCache*.

Communication of polygons across different reduce tasks (during shuffle and sort phase) has its own overhead. This algorithm is geared towards processing overlay operation locally in map phase itself. Instead of computing overlay in reduce phase, a combiner phase can be used for local computation in map phase itself. Instead, we implement In-Mapper Combining design pattern [106] by using *cleanup* method. The usage of this design pattern reduces the intermediate *key-value* pairs and it is shown to perform better than using a combiner phase. Since *cleanup* is called after all the base and overlay layer polygons are read by a mapper, polygon overlay can be locally computed in a distributed fashion.

Let us consider an example with four base layer polygons (b1 to b4) and five overlay layer polygons (c1 to c5) to illustrate the algorithm with regard to the communication of polygons in map, shuffle and reduce phases of this algorithm. A base layer polygon can potentially overlay with zero or more overlay layer polygons and this overlap relationship is shown in Table 6.1 as an intersection graph represented as an adjacency list. Figure 6.1 illustrates the

whole data flow happening in the second job of the Algorithm 8 for the polygons mentioned in Table 6.1. In figure 6.1, we can see how the overlapping polygons from two different layers get grouped together after the map and shuffle phases. In the figure, as we can see, two overlapping polygons b1 and c1 are locally present in the first input file split and as such are overlaid in the *cleanup* method itself. Moreover, as we can see in the intersection graph, overlay layer polygon c2 is not emitted out for further overlay processing in the map phase of the second job since it does not overlap with any of the base layers polygons.

Table (6.1) *Intersection graph represented as an adjacency list.*

Base layer polygon	**Overlay layer polygon**
b1	c1, c3
b2	c4, c5
b3	c1
b4	-

The advantage of having a local R-tree in all the mapper tasks is that each mapper can independently determine those overlay layer polygons which do not overlap with any base layer polygon and as such can be discarded thereby preventing such polygons from going to shuffle and reduce phase. Moreover, in a reducer with *key-value* pair $< k, v >$, for a given key k (base polygon id), the value v consists of only those overlay layer polygons that can potentially intersect with the base polygon with id k.

6.3.4 Overlay Algorithm with Map Phase Only using *DistributedCache*

In some scenarios, we can exploit an optimization when one of the input file is small enough to fit entirely in memory by using *DistributedCache*. The optimization results from the fact that data transfer overhead involved in transferring key-value pairs from mappers to reducers is avoided. In some real-world GIS applications, when a small base layer file

Algorithm 8 Chained Map Reduce Algorithm

JOB I
method MAP (*id*, polygon *p*)
{
if p is from base layer **then**
  extract bounding box of p
  EMIT: (id_p, bounding box of *p*)
end if
}

JOB II
method INITIALIZE
{
Initialize R-tree
Read bounding boxes of base layer polygons from *DistributedCache*
Insert bounding boxes to R-tree
}

method MAP (polygon p)
{
if *p* is from base layer **then**
  extract *id* from *p* and store in list *B*
  EMIT: (*id*, *p*)
else
  parse *p* and store in a list *C*
end if
}

method CLEANUP (list *B*, list *C*)
{
for all overlay polygon $p_c \in C$ **do**
  get intersecting polygon id list *L* from R-tree
  for all base polygon $p_b \in L$ **do**
    if p_b is present locally in *B* **then**
      compute overlay (p_b, p_c)
      EMIT: output polygon
    else
      EMIT: (id_b, p_c)
    end if
  end for
end for
}

method REDUCE (base polygon id, [p_1, p_2, ..])
{
extract base layer polygon p_b from [p_1, p_2, ..]
for each overlay layer polygon p_c in [p_1, p_2, ..]
  compute overlay (p_b, p_c)
  EMIT: output polygon
}

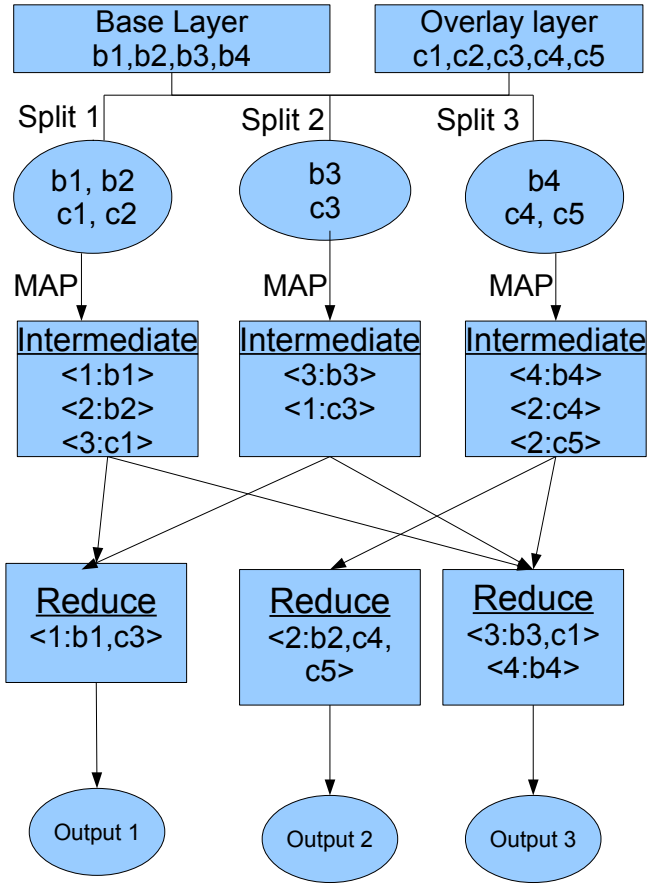

Figure (6.1) *Illustration of Job II of Chained MapReduce Overlay Algorithm.*

has to be overlaid on top of larger overlay layer file, then this base layer file can be simply replicated on all the machines. In Algorithm 9, the base layer file is broadcasted to all the slave nodes. This algorithm has memory limitation since base layer polygons needs to be in-memory.

Each mapper loads the base layer file into its local memory and calls the map function for each tuple. It should be noted that the base layer file is not partitioned among the slave nodes as a result of which individual map tasks in this algorithm lose much of the advantage of data locality. But, overall job gains due to the potential for the elimination of reduce

phase along with the elimination of the shuffle and sort after map phase. For the sake of efficient query, spatial index is created using quadtree or R-tree. But, spatial index creation has its own overhead. This algorithm avoids the overhead of spatial index creation and relies on sorting of polygons to find potentially intersecting polygons.

Algorithm 9 *DistributedCache* Based Overlay Algorithm

```
method INITIALIZE
{
Read base polygon list B from the DistributedCache
Sort B on x-coordinate of bottom-left corner of bounding box of polygons
}

method MAP (id, polygon p)
{
if p is from overlay layer then
  x ← x-coordinate of top-right corner of bounding box of p
  Index n ← BinarySearch (B, x)
  L ← {b_0, b_1, .. , b_n}, b_i ∈ B
  for all polygon b_i ∈ L do
    compute overlay (b_i, p)
    EMIT: output polygon
  end for
end if
}
```

6.3.5 Grid-based Overlay Algorithm with a single Map and Reduce phase

In the Algorithm 8, every mapper creates a complete R-tree redundantly. The disadvantage of this approach is that for very large number of polygons, there may be memory constraints. Here, we partition the polygonal data in a cellular grid in such a way that each cell of the grid contains all the polygons that lies in it partially or completely. It is possible that some of the polygons may span across multiple grid cells. Now, in each cell, we can build an R-tree from the polygons present locally. In this way, an R-tree can be built in each partition in a top-down fashion starting from the root by inserting the bounding boxes iteratively. It should be noted that this is not a distributed R-tree construction. The benefit

of doing this is that instead of having a global R-tree, now we can have a local and smaller R-tree in each grid cell which can be efficiently queried. An R-tree built in this manner is suitable for local overlay processing in each grid cell since all of the potentially overlapping spatial features will be already indexed and as such can be efficiently queried for overlaying polygons. Algorithm 10 shows the map and reduce phases.

Initially, the dimension of the grid is determined which is a minimum bounding box spatially containing all the polygons from base layer and overlay layer. Then, the dimension of grid cells is computed based on the number of partitions. The number of cells should be greater than the reduce capacity of the cluster in order to ensure proper load-balancing. A polygon may belong to one or more cells and since we know the bounding box of all the grid cells, each mapper task can independently determine to which cell(s) a polygon belongs to. After all the mappers are done, each reducer gets a subset of base layer and overlay layer polygons. Then, a local R-tree is built from the overlay layer polygons and for every base layer polygon, the intersecting overlay layer polygons are found out by querying the R-tree. Finally, the overlay operation is carried out using clipper library and output polygons are written to HDFS.

6.4 Experimental Setup

The cluster used in the experiments is a heterogenous linux cluster containing (i) four nodes (model AMD Quad Core Opteron 2376, 2.3 GHz) with 8 CPU cores, ii) four nodes (model Intel Xeon Quad Core 5410, 2.33 GHz) with 8 CPU cores, (iii) one node (model AMD Quad-Core Opteron Processor 8350, 2.0 GHz) with 16 CPU cores, (iv) one node (model AMD Quad-Core Opteron Processor 8350, 2.1 GHz) with 32 CPU cores and (v) one node (model AMD Opteron Processor 6272, 2.1 GHz) with 64 CPU cores interconnected by an InfiniBand network. In our cluster all the nodes share same file system hosted at the head node. We have installed Apache's Hadoop version 1.02, which is an open source implementation of the MapReduce programming model. HDFS has a master/slave architecture consisting of a namenode which acts as a master server that manages the file system namespace and a

Algorithm 10 Grid based Overlay Algorithm with Map and Reduce Phase

```
method MAP (id, polygon p)
{
compute bounding box b of p
find overlapping grid cell(s) C for p based on b
for all grid cell ∈ C do
   EMIT: (cell id, p)
end for
}

method REDUCE (cell id, [p1, p2, .. ])
{
for all  p in [p1, p2, .. ] do
   if p is from base layer then
      add to list B
   else
      compute the bounding box b of p
      add b to R-tree tree
   end if
end for

for each base polygon pb in B
   compute bounding box b of pb
   C ← get overlapping polygons by searching for b in tree
   for each overlay layer polygon pc in C
      compute overlay (pb, pc)
      EMIT: output polygon
}
```

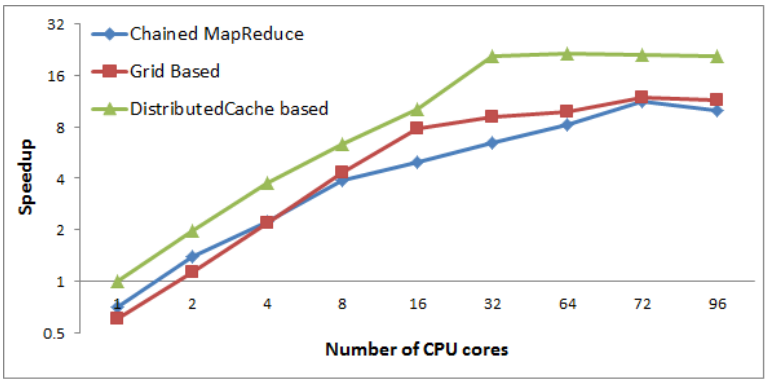

Figure (6.2) *Performance impact of varying number of CPU cores for overlay algorithms using Dataset 1.*

number of datanodes that manage storage attached to the nodes. One of the node with 8 CPU cores is configured for running namenode, secondary namenode node and job tracker. The rest of the compute nodes are configured for running datanodes and tasktrackers. We performed timing experiments with real world data sets as shown in Table 6.2.

Table (6.2) *Description of data sets used in experiments.*

Data set	**Map layer**	**Size (MB)**	**Number of polygons**
Dataset 1	Base layer	32	4000
	Overlay layer	2300	1000000
Dataset 2	Base layer	550	300000
	Overlay layer	770	750000

Figure 6.2 and 6.3 shows the speedup of the three versions of overlay algorithm for dataset 1 and dataset 2. All of the three overlay algorithms show better speedup for dataset 1 in comparison to the dataset 2. For the overlay algorithm with a map phase only (referred as *DistributedCache* based algorithm in the figures), only one of the input layer is partitioned across the mappers and the other layer is entirely in *DistributedCache*, as such parallelism is

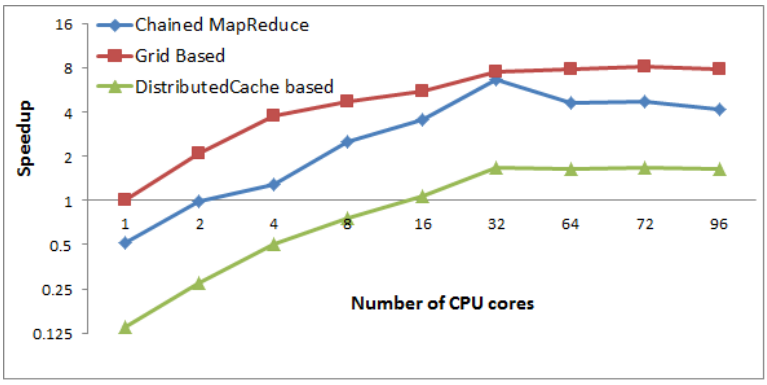

Figure (6.3) *Performance impact of varying number of CPU cores for overlay algorithms using Dataset 2.*

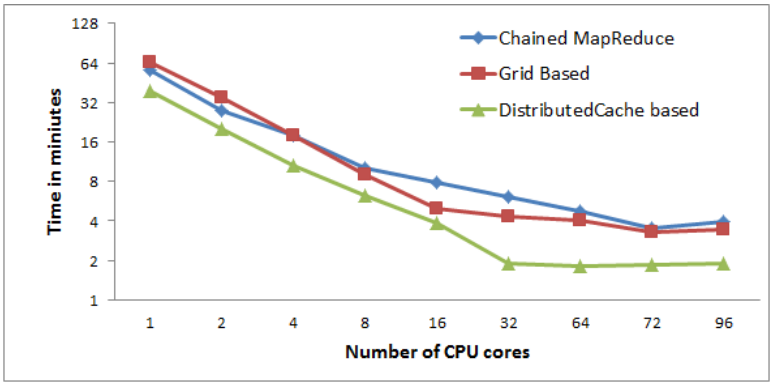

Figure (6.4) *Execution time for different overlay algorithms using Dataset 1.*

limited resulting in speedup getting saturated upto 32 cores only. The amount of parallelism available in the *DistributedCache* based algorithm is directly proportional to the number of mappers that can concurrently run in the cluster. The number of mappers is equivalent to the number of partitions (input splits) of the input file which is thirty six ((2300 + 32)/ 64) in case of dataset 1, and twelve ((770 + 550)/64) in case of dataset 2. As such the dataset 1 shows better speedup in comparison to the dataset 2.

Figure 6.4 and 6.5 are the timing plots for different algorithms using dataset 1 and dataset 2 respectively which shows the total time taken to complete the overlay operation as the number of CPU cores increase. In both figures, the time taken to process the dataset 1 is

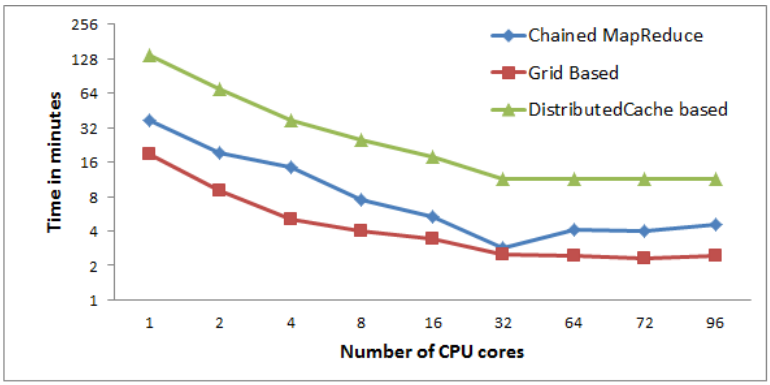

Figure (6.5) *Execution time for different overlay algorithms using Dataset 2.*

greater than the dataset 2 for Chained MapReduce overlay algorithm and Grid-based overlay algorithm with Map and Reduce phases. On the other hand, for *DistributedCache* based algorithm, the time taken to process dataset 2 is greater than the dataset 1. This anomalous behaviour can be explained by the cost of accessing larger base layer file (550 MB) using *DistributedCache* in case of dataset 2. On the other hand, in case of dataset 1, only a 32 MB file is in-memory in all the mappers which is seventeen times smaller than the size of the base layer file in the dataset 2. Thus, we observed that an algorithm with only a map phase using *DistributedCache* is suitable for those applications where a relatively smaller file has to be stored in the *DistributedCache*. Although more experiments are required to predict exactly the performance impact of file sizes in *DistributedCache*, we observed noticeable performance degradation in case of larger files.

Figure 6.6 and 6.7 shows the breakdown of average execution times for map, shuffle and reduce phases of Chained MapReduce algorithm for the two data sets. Figure 6.8 and 6.9 shows the breakdown of average execution time for map, shuffle and reduce phases of grid-based overlay algorithm for the two data sets. As we can see from the above mentioned figures, the reduce phase is the most time consuming where the bulk of overlay computation actually takes place. Map phase consumes the least time of all the three phases. Map phase for Chained MapReduce algorithm performs local overlay computation and as such takes

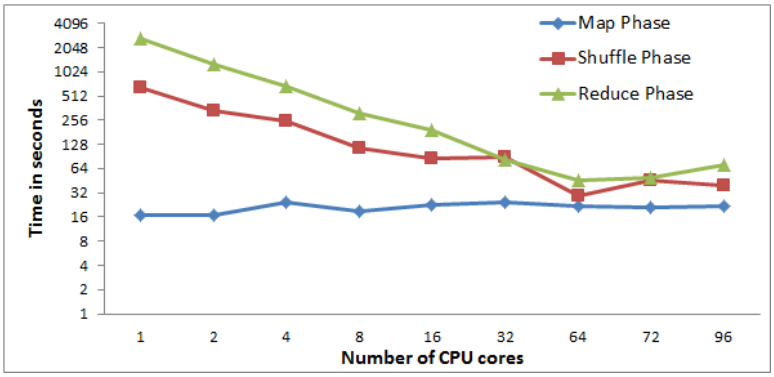

Figure (6.6) *Average execution time for different phases in Chained MapReduce Overlay Algorithm using Dataset 1.*

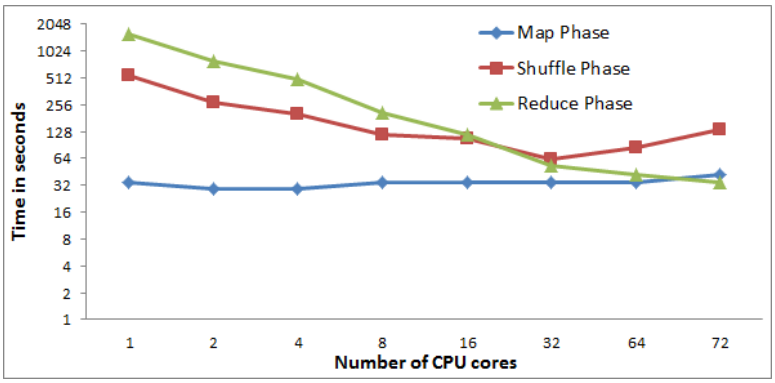

Figure (6.7) *Average execution time for different phases in Chained MapReduce Overlay Algorithm using Dataset 2.*

slightly more time than the map phase of grid-based algorithm where there is no overlay computation in map phase. Currently, we have performed experiments with small data sets (less than 2.5 GB) and we observed that the scalability is limited. It will be interesting to see the scalability with larger data sets. The scalability may be improved by applying dynamic load balancing techniques in the existing algorithms.

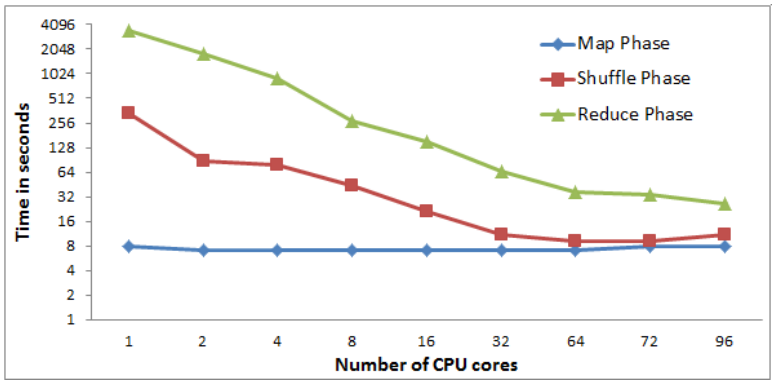

Figure (6.8) *Average execution time for different phases in Grid-based Overlay Algorithm with Map and Reduce phases using Dataset 1.*

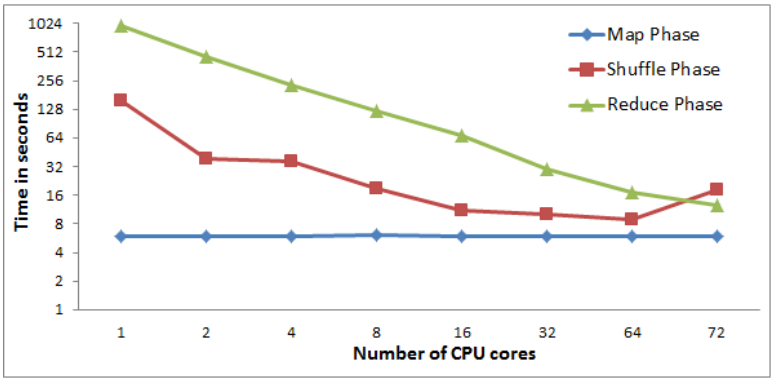

Figure (6.9) *Average execution time for different phases in Grid-based Overlay Algorithm with Map and Reduce phases using Dataset 2.*

PART 7

CONCLUSIONS

Although polygon clipping and related problems have been researched since late seventies, the best algorithm [40] was not output sensitive and did not handle arbitrary polygons. Our PRAM algorithm is developed from the first principles and it shows the parallelization of a plane-sweep based algorithm relying only on parallel primitives such as prefix sum and sorting. We tested our multi-threaded algorithms with real-world and synthetic datasets and achieved 30x speedup using a 64-core AMD Opteron Processor. We have discussed in detail about the intersection operation but it can be easily extended to other operations for example union, difference, etc., by simply changing the vertex labeling rules.

In this dissertation, we have developed an efficient parallel GH algorithm for clipping a pair of polygons. The cost of our algorithm matches the time complexity of sequential sweepline based polygon clipping. This is also an improvement over our previous parallel clipping algorithm which required $O(k')$ additional processors, where k' is the number of extra vertices introduced due to partitioning of polygons. We have also implemented multi-threaded GH algorithm using CUDA on GPU which is faster than multi-core implementation of GH algorithm and sequential sweepline algorithm for larger polygons.

We have developed and implemented *MPI-GIS* which is a distributed polygon overlay processing system for real-world polygonal datasets on a linux cluster. We plan to include more compute-intensive geometric algorithms like buffer computation, Minkowski sum, etc. We believe that *MPI-GIS* can be extended by integrating it with GPU based parallel polygon clipping and employing a parallel file system such as Lustre.

We have also experimented with three MapReduce algorithms (excluding naive algorithm) using two data sets and the performance of these algorithms depend on the size and nature of the data. Even though, we have discussed polygon overlay in particular, some of

the techniques discussed in this paper (related to use of *DistributedCache* and In-Mapper combining design pattern) are applicable for parallelizing similar binary applications using MapReduce framework.

REFERENCES

[1] H. Alt and L. Scharf, "Computation of the hausdorff distance between sets of line segments in parallel," *arXiv preprint arXiv:1207.3962*, 2012.

[2] B. Vatti, "A generic solution to polygon clipping," *Communications of the ACM*, vol. 35, no. 7, pp. 56–63, 1992.

[3] 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.

[4] L. Becker, A. Giesen, K. H. Hinrichs, and J. Vahrenhold, "Algorithms for performing polygonal map overlay and spatial join on massive data sets," in *Advances in spatial databases*. Springer, 1999, pp. 270–285.

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

[6] P. Martens, G. Attrill, A. Davey, A. Engell, S. Farid, P. Grigis, J. Kasper, K. Korreck, S. Saar, A. Savcheva, Y. Su, P. Testa, M. Wills-Davey, P. Bernasconi, N.-E. Raouafi, V. Delouille, J. Hochedez, J. Cirtain, C. DeForest, R. Angryk, I. De Moortel, T. Wiegelmann, M. Georgoulis, R. McAteer, and R. Timmons, "Computer vision for the solar dynamics observatory (sdo)," in *The Solar Dynamics Observatory*, P. Chamberlin, W. Pesnell, and B. Thompson, Eds. Springer US, 2012, pp. 79–113. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-3673-7_6

[7] S. Prasad, S. Shekhar, S. Puri, M. McDermott, D. Agarwal, S. Karamati, X. Zhou, and M. Evans, "Hpc-geo: High performance computing over geo-spatiotemporal data: A summary of results. (position paper)," in *Procs. The All Hands Meeting of the NSF CyberGIS project*, 2014.

[8] S. Puri and S. K. Prasad, "Parallel Algorithm for Clipping Polygons with improved bounds and A Distributed Overlay Processing System using MPI," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015 (accepted).

[9] S. Ray, B. Simion, A. D. Brown, and R. Johnson, "A parallel spatial data analysis infrastructure for the cloud," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2013, pp. 274–283.

[10] R. G. Healey, S. Dowers, and M. Minetar, *Parallel processing and GIS*. Taylor & Francis, Inc., 1996.

[11] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Parallel processing of spatial joins using r-trees," in *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*. IEEE, 1996, pp. 258–265.

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

[13] Open Topography Facility, "Open topography," http://opentopo.sdsc.edu/gridsphere/gridsphere?cid

[14] D. Agarwal, S. Puri, X. He, and S. K. Prasad, "Cloud computing for fundamental spatial operations on polygonal GIS data," *Cloud Futures*, 2012.

[15] D. Agarwal and S. K. Prasad, "Lessons learnt from the development of GIS application on azure cloud platform," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 352–359.

[16] S. Puri, D. Agarwal, X. He, and S. K. Prasad, "MapReduce algorithms for GIS Polygonal Overlay Processing," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2013.

[17] S. Puri and S. K. Prasad, "Efficient parallel and distributed algorithms for GIS polygonal overlay processing," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE Computer Society, 2013, pp. 2238–2241.

[18] 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," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[19] SpatialHadoop, http://spatialhadoop.cs.umn.edu. Website. [Online]. Available: http://spatialhadoop.cs.umn.edu/

[20] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan, "CG_Hadoop: Computational geometry in MapReduce," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2013, pp. 284–293.

[21] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "Sjmr: Parallelizing spatial join with mapreduce on clusters," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–8.

[22] 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.

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

[24] S.-H. Lo, C.-R. Lee, Y.-C. Chung, and I.-H. Chung, "A parallel rectangle intersection algorithm on gpu+ cpu," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE, 2011, pp. 43–52.

[25] S.-H. Lo, C.-R. Lee, I. Chung, Y.-C. Chung *et al.*, "Optimizing pairwise box intersection checking on gpus for large-scale simulations," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 23, no. 3, p. 19, 2013.

[26] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz, "Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1543–1554, 2012.

[27] M. McKenney, G. De Luna, S. Hill, and L. Lowell, "Geospatial overlay computation on the gpu," in *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2011, pp. 473–476.

[28] S. Audet, C. Albertsson, M. Murase, and A. Asahara, "Robust and efficient polygon overlay on parallel stream processors," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2013, pp. 294–303.

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

[30] G. Greiner and K. Hormann, "Efficient clipping of arbitrary polygons," *ACM Transactions on Graphics (TOG)*, vol. 17, no. 2, pp. 71–83, 1998.

[31] T. Yampaka and P. Chongstitvatana, "Spatial join with R-tree on graphics processing units, IC2IT," 2012.

[32] C. Sun, D. Agrawal, and A. El Abbadi, "Hardware acceleration for spatial selections and joins," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 455–466.

[33] J. Zhang and S. You, "Speeding up large-scale point-in-polygon test based spatial join

on GPUs," in *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. ACM, 2012, pp. 23–32.

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

[35] A. Aji, G. Teodoro, and F. Wang, "Haggis: Turbocharge a MapReduce based spatial data warehousing system with GPU engine," in *Proceedings of the ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. ACM, 2014.

[36] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[37] A. Danner, A. Breslow, J. Baskin, and D. Wilikofsky, "Hybrid MPI/GPU interpolation for grid DEM construction," in *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. ACM, 2012, pp. 299–308.

[38] C. Lai, M. Huang, X. Shi, and H. You, "Accelerating geospatial applications on hybrid architectures," *IEEE International Conference on High Performance Computing and Communications*, 2013.

[39] S. K. Prasad, S. Shekhar, X. Zhou, M. McDermott, S. Puri, D. Shah, and D. Aghajarian, "A Vision for GPU-accelerated Parallel Computation on Geo-Spatial Datasets," in *Sigspatial Newsletter Special issue on Big Spatial Data*, 2014 (to appear in.).

[40] R. Karinthi, K. Srinivas, and G. Almasi, "A parallel algorithm for computing polygon set operations," in *Parallel Processing Symposium, 1994. Proceedings., Eighth International*. IEEE, 1994, pp. 115–119.

[41] C. Rüb, "Line segment intersection reporting in parallel," *Algorithmica*, vol. 8, no. 1, pp. 119–144, 1992.

[42] M. J. Atallah and M. T. Goodrich, "Efficient plane sweeping in parallel," in *Proceedings of the Second Annual Symposium on Computational geometry*. ACM, 1986, pp. 216–225.

[43] M. De Berg, O. Cheong, and M. Van Kreveld, *Computational geometry: algorithms and applications*. Springer, 2008.

[44] C. Rub, "Computing intersections and arrangements for red-blue curve segments in parallel." *Proceedings of the Fourth Canadian Conference on Computational Geometry*, 1992.

[45] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," *SIAM Journal on Computing*, vol. 20, no. 4, pp. 737–755, 1991.

[46] I. E. Sutherland and G. W. Hodgman, "Reentrant polygon clipping," *Communications of the ACM*, vol. 17, no. 1, pp. 32–42, 1974.

[47] Y.-D. Liang and B. A. Barsky, "An analysis and algorithm for polygon clipping," *Communications of the ACM*, vol. 26, no. 11, pp. 868–877, 1983.

[48] B.-O. Schneider and J. van Welzen, "Efficient polygon clipping for an SIMD graphics pipeline," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 4, no. 3, pp. 272–285, 1998.

[49] M. Qatawneh, A. Sleit, and W. Almobaideen, "Parallel implementation of polygon clipping using transputer," *American Journal of Applied Sciences*, vol. 6, no. 2, p. 214, 2009.

[50] C. Narayanaswami, "A parallel polygon-clipping algorithm," *The Visual Computer*, vol. 12, no. 3, pp. 147–158, 1996.

[51] T. Theoharis and I. Page, "Two parallel methods for polygon clipping," in *Computer graphics forum*, vol. 8, no. 2. Wiley Online Library, 1989, pp. 107–114.

[52] F. Martinez, A. J. Rueda, and F. R. Feito, "A new algorithm for computing boolean operations on polygons," *Computers & Geosciences*, vol. 35, no. 6, pp. 1177–1185, 2009.

[53] J. Nievergelt and F. P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Communications of the ACM*, vol. 25, no. 10, pp. 739–747, 1982.

[54] M. McKenney and T. McGuire, "A parallel plane sweep algorithm for multi-core systems," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2009, pp. 392–395.

[55] A. B. Khlopotine, V. Jandhyala, and D. Kirkpatrick, "A variant of parallel plane sweep algorithm for multicore systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 6, pp. 966–970, 2013.

[56] C. Narayanaswami and W. R. Franklin, "Boolean combinations of polygons in parallel," in *Proceedings of the 1992 International Conference on Parallel Processing*, vol. 3, 1992, pp. 131–135.

[57] F. Dehne and J.-R. Sack, "A survey of parallel computational geometry algorithms," in *Parcella'88*. Springer, 1989, pp. 73–88.

[58] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel computational geometry," *Algorithmica*, vol. 3, no. 1-4, pp. 293–327, 1988.

[59] G. Greiner and K. Hormann, "Efficient clipping of arbitrary polygons," *ACM Transactions on Graphics (TOG)*, vol. 17, no. 2, pp. 71–83, 1998.

[60] S. G. Akl, *Parallel computation: models and methods*. Prentice-Hall, Inc., 1997.

[61] D. M. Mount, "Geometric intersection," in *Handbook of Discrete and Computational Geometry, chapter 33*. Citeseer, 1997.

[62] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

[63] M. Agoston, "Vatti polygon clipping," in *Computer Graphics and Geometric Modelling*. Springer, 2005, pp. 98–105.

[64] GPC Clipper library. [Online]. Available: http://www.cs.man.ac.uk/~toby/alan/software/gpc.html

[65] B. R. Vatti, "A generic solution to polygon clipping," *Commun. ACM*, vol. 35, pp. 56–63, July 1992. [Online]. Available: http://doi.acm.org/10.1145/129902.129906

[66] Java Topology Suite. [Online]. Available: http://www.vividsolutions.com/jts/JTSHome.htm

[67] GML datasets. [Online]. Available: http://www.cs.gsu.edu/~dimos/?q=content/parallel-and-distributed-algorithms-polygon-clipping.html

[68] R. Karinthi, K. Srinivas, and G. Almasi, "A parallel algorithm for computing polygon set operations," *Journal of Parallel and Distributed Computing*, vol. 26, no. 1, pp. 85–98, 1995.

[69] S. Puri and S. K. Prasad, "Output-sensitive parallel algorithm for polygon clipping," in *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 2014, pp. 241–250.

[70] "Hazus," Website, 2013. [Online]. Available: http://www.fema.gov/plan/prevent/hazus/

[71] 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 *IPDPS Workshops*, 2012, pp. 1433–1439.

[72] NERSC Carver cluster. Website. [Online]. Available: http://www.nersc.gov/users/computational-systems/carver/

[73] F. L. Dévai, "On the complexity of some geometric intersection problems," in *Journal of Computing and Information*. Citeseer, 1995.

[74] PNPOLY Point-In-Polygon test. Website. [Online]. Available: http://www.ecse.rpi.edu/~wrf/Research/Short_Notes/pnpoly.html

[75] S. G. Akl, *Parallel computation: models and methods*. Prentice-Hall, Inc., 1997.

[76] Jcuda library. Website. [Online]. Available: http://www.jcuda.org/

[77] Classic Polygon dataset. Website. [Online]. Available: http://www.complex-a5.ru/polyboolean/downloads.html

[78] V. Solutions, "Java Topology Suite," 2003.

[79] Alan Murta, "General Polygon Clipper library," Website, 2012. [Online]. Available: http://www.cs.man.ac.uk/~toby/gpc/

[80] Hazus website. [Online]. Available: http://www.fema.gov/plan/prevent/hazus/

[81] J. D. Hobby, "Practical segment intersection with finite precision output," *Computational Geometry*, vol. 13, no. 4, pp. 199 – 214, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925772199000218

[82] R. G. Healey, M. J. Minetar, and S. Dowers, Eds., *Parallel Processing Algorithms for GIS*. Bristol, PA, USA: Taylor & Francis, Inc., 1997.

[83] F. Wang, "A parallel intersection algorithm for vector polygon overlay," *Computer Graphics and Applications, IEEE*, vol. 13, no. 2, pp. 74 –81, mar 1993.

[84] T. Waugh and S. Hopkins, "An algorithm for polygon overlay using cooperative parallel processing," *International Journal of Geographical Information Science*, vol. 6, no. 6, pp. 457–467, 1992.

[85] S. Hopkins and R. Healey, "A parallel implementation of franklins uniform grid technique for line intersection detection on a large transputer array," *Brassel and Kishimoto [BK90]*, pp. 95–104, 1990.

[86] Q. Zhou, E. Zhong, and Y. Huang, "A parallel line segment intersection strategy based on uniform grids," *Geo-Spatial Information Science*, vol. 12, no. 4, pp. 257–264, 2009.

[87] M. Armstrong and P. Densham, "Domain decomposition for parallel processing of spatial problems," *Computers, environment and urban systems*, vol. 16, no. 6, pp. 497–513, 1992.

[88] P. K. Agarwal, L. Arge, T. Mølhave, and B. Sadri, "I/O-efficient Efficient Algorithms for Computing Contours on A Terrain," in *Symposium on Computational Geometry*, 2008, pp. 129–138.

[89] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha, "Fast Computation of Database Operations using Graphics Processors," in *SIGMOD Conference*, 2004, pp. 215–226.

[90] T. M. Chan, "A simple trapezoid sweep algorithm for reporting red/blue segment intersections," in *In Proc. 6th Canad. Conf. Comput. Geom*, 1994, pp. 263–268.

[91] B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting line segments in the plane," *J. ACM*, vol. 39, pp. 1–54, January 1992. [Online]. Available: http://doi.acm.org/10.1145/147508.147511

[92] S. Dowers, B. M. Gittings, and M. J. Mineter, "Towards a framework for high-performance geocomputation: handling vector-topology within a distributed service environment," *Computers, Environment and Urban Systems*, vol. 24, no. 5, pp. 471 – 486, 2000.

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

[94] Crayons - an azure cloud based parallel system for gis overlay operations. [Online]. Available: www.cs.gsu.edu/dimos/crayons.html

[95] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD Conference*, 1984, pp. 47–57.

[96] E. ESRI, "Shapefile technical description," *Environmental Systems Research Institute, Inc*, 1998.

[97] Angus Johnson, "Polygon Clipper library," Website, 2014. [Online]. Available: http://sourceforge.net/projects/polyclipping/files/

[98] J.-P. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," in *Data Engineering, 2000. Proceedings. 16th International Conference on.* IEEE, 2000, pp. 535–546.

[99] S. Audet, C. Albertsson, M. Murase, and A. Asahara, "Robust and efficient polygon overlay on parallel stream processors," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 2013, pp. 294–303.

[100] 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 *IEEE International Parallel and Distributed Processing Symposium workshops*, 2012.

[101] H. Langendoen, "Parallelizing the polygon overlay problem using Orca," *Student Project Report, Vrije Universiteit Amsterdam*, 1995.

[102] J. Zhao, Q. Li, and H. Zhou, "A cloud-based system for spatial analysis service," in *Remote Sensing, Environment and Transportation Engineering (RSETE), 2011 International Conference on.* IEEE, 2011, pp. 1–4.

[103] Y. Liu, N. Jing, L. Chen, and H. Chen, "Parallel bulk-loading of spatial data with mapreduce: An r-tree case," *Wuhan University Journal of Natural Sciences*, vol. 16, no. 6, pp. 513–519, 2011.

[104] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," in *Scientific and Statistical Database Management*. Springer, 2009, pp. 302–319.

[105] D. Agarwal and S. K. Prasad, "Lessons learned from the development of GIS overlay processing application on azure cloud platform," 2012.

[106] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. ACM, 2010, pp. 78–85.